Requirements Interaction Management

WILLIAM N. ROBINSON, SUZANNE D. PAWLOWSKI, AND VECHESLAV VOLKOV

Georgia State University

Requirements interaction management (RIM) is the set of activities directed toward the discovery, management, and disposition of critical relationships among sets of requirements, which has become a critical area of requirements engineering. This survey looks at the evolution of supporting concepts and their related literature, presents an issues-based framework for reviewing processes and products, and applies the framework in a review of RIM state-of-the-art. Finally, it presents seven research projects that exemplify this emerging discipline.

Categories and Subject Descriptors: C.0 [**General**]: *System architectures; system specification methodology*; C.4 [**Performance of Systems**]: *Modeling techniques; performance attributes*; D.2.1 [**Software Engineering**]: *Requirements/Specifications*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.4 [**Software Engineering**]: Software Program Verification; D.2.9 [**Software Engineering**]: Management—*Life cycle; software quality assurance (SQA)*; D.2.10 [**Software Engineering**]: Design; H.1.1 [**Models and Principles**]: Systems and Information Theory; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms: Design, Management, Performance, Reliability, Verification

Additional Key Words and Phrases: Requirements engineering, system specification, system architecture, analysis and design, dependency analysis, interaction analysis, composite system, WinWin, Telos, distributed intentionality, viewpoints, KAOS, deficiency driven design, KATE, Oz, software cost reduction (SCR).

1. INTRODUCTION

One of the main objectives of requirements engineering (RE) is to improve systems modeling and analysis capabilities so that organizations can better understand critical system aspects before they actually build the system. As Brooks [1987] noted, requirements definition is the most difficult development stage:

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements.... No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later. [Brooks 1987, p. 18]

Consequently, requirements engineering research spans a wide range of topics [Pohl 1997], but a topic of increasing importance is the analysis and management of dependencies among requirements. We call this *requirements interaction management* (*RIM*) and define it as "the set of activities directed toward

Authors' address: Department of Computer Information Systems, Georgia State University, Atlanta, GA 30302; email: {wrobinson,spawlowski}@gsu.edu;vvolkov@earthlink.com.

the discovery, management, and disposition of critical relationships among sets of requirements." Although the term itself is new, requirements engineers have long recognized that the topics and issues RIM encompasses are crucial to obtaining a good requirements specification.

The thrust of RIM is to analyze the extent to which a system can satisfy multiple requirements simultaneously. A system has many components, and each component has many requirements—requirements that can interact with other requirements and with the environment. The satisfaction of one requirement can aid or detract from the satisfaction of another, and the environment can increase or reduce requirement satisfaction [van Lamsweerde and Letier 2000]. Single-requirement methods exist—for example, to minimize network latency or maximize network throughput—but they typically apply to only one or a few requirements. As object-oriented methods and networked system deployment become more common, the number of components (and their interactions) will increase. Satisfying all these requirements through component composition becomes extremely difficult.

As Neumann [1995] has suggested,

> The satisfaction of a single requirement is difficult enough, but the simultaneous and continued satisfaction of diverse and possibly conflicting requirements is typically much more difficult. [Neumann 1995, p. 2]

Yet, despite the importance of managing requirements interactions, the state of the art suffers three major problems [van Lamsweerde et al. 1998]:

(1) The specific kind of interaction being considered is not always clear.

(2) There is a lack of systematic techniques for detecting conflicts among nonoperational requirements.

(3) There is a lack of systematic techniques for resolving conflicts.

RIM seeks to address all three.

1.1. Problematic Interaction

Component interaction errors, which arise from incorrect requirements, have become a significant development problem. Requirements errors are *numerous*: they typically make up 25% to 70% of total software errors—US companies average one requirements error per function point [Jones 1995]. They can be *persistent*: two-thirds are detected after delivery. They can be *expensive*: the cost to fix them can be up to a third of the total production cost [Boehm 1981]. Moreover, many system failures are attributed to poor requirements analysis [Jones 1996; Lyytinen and Hirschheim 1987; Neumann 1995].

Component interaction errors can be more serious than simple component failures [Perrow 1984]. As Leveson [1995] observed:

> Whereas in the past, component failure was cited as the major factor in accidents, today more accidents result from dangerous design characteristics and interaction among components [Hammer 1980]. [Leveson 1995, p. 9]

Leveson documented several cases in which incorrect requirements caused component interaction errors with grave consequences.

1.2. Understanding Requirements Conflict

At first glance, it may appear straightforward to support requirements interaction analysis: simply formalize the requirements, or at least structure them, and then use a computer-aided software engineering (CASE) tool to check syntax and consistency. However, although CASE tools have successfully provided support for modeling and code generation [Chikofsky and Rubenstein 1993; Lempp and Rudolf 1993; Norman and Nunamaker 1989], they have been less successful in supporting requirements analysis [Lempp and Rudolf 1993]. (In fact, the downstream life-cycle successes of these tools may be one reason systems analysts are spending increasing amounts of time on requirements analysis [Graf and Misic 1994].) Moreover, requirements analysis is not just about checking the consistency of

descriptions. In fact, inconsistent requirements often reflect the inconsistent needs of system stakeholders—something developers need to see. Inconsistent requirements are the starting point for deriving useful information that might otherwise go unnoticed [Finkelstein et al. 1994].

Inconsistency [Nuseibeh et al. 1994], *conflict, breakdown* [Winograd and Flores 1987], *cognitive dissonance* [Festinger 1964]—all are terms that characterize aspects of uncovering unexpected ideas during problem solving and they are common in requirements engineering literature (see Section 4.2.1). Conflict is an important driver of group communication and productive work [Robbins 1983], and research has shown empirically that it drives systems development [Lyytinen and Hirschheim 1987; Markus and Keil 1994; Robey et al. 1989] and, more specifically, requirements development [Bendifallah and Scacchi 1989; Kim and Lee 1986; Magal and Snead 1993; Robinson 1990].

Two basic forces give rise to requirements conflict. First, the technical nature of constructing a requirements document gives rise to *inconsistency*—"any situation in which two parts of a [requirements] specification do not obey some relationship that should hold between them" [Easterbrook and Nuseibeh 1996, p. 32]. Second, the social nature of constructing a requirements document gives rise to *conflict*—requirements held by two or more stakeholders that cause an inconsistency. Applying the general convention in requirements engineering, we use the term *conflict* to indicate both problems unless the context calls for the use of a more specific term.

Consider three *technical difficulties* that lead to requirements conflict:

—*Voluminous requirements*. The sheer size of a requirements document can lead to conflicts, such as varied use of terminology. This is especially true when multiple analysts elaborate the requirements.

—*Changing requirements and analysts*. As a requirements document evolves, developers add new requirements and update older ones. One change request can lead to a cascade of other change requests until the requirements reach a consistent state. Consequently, the document is typically in a transitory state with many semantic conflicts. Requirements analysts expect to resolve most of these by bringing them to their current state (as the analysts interpret "current state"). Unfortunately, the implicit current state of requirements is lost when analysts leave a long-term project. Moreover, requirements concepts and their expressions vary with the development team composition.

—*Complex requirements*. The complexity of the domain or software specification can make it difficult to understand exactly what has been specified or how components interact. If the development team and stakeholders are struggling to understand the requirements naturally, they are less likely to see any requirements dependencies and thus likely to overlook requirements conflicts.

Consider three *social difficulties* that lead to requirements conflict:

—*Conflicting stakeholder requirements*. Different stakeholders often seek different requirements that the system cannot satisfy together. For example, one stakeholder might want to use an open-source communication protocol, while another wants a proprietary solution.

—*Changing and unidentified stakeholders*. In the attempt to understand system requirements, analysts often seek new stakeholders for an ongoing project. Analysts report that they can understand system requirements when interacting with actual users, but that it is difficult to gain access to them [Lubars et al. 1993]. Moreover, one department of an organization may claim to be "the" customer, but another department may make the final purchasing decision [Lubars et al. 1993]. Thus, a previously unidentified stakeholder becomes an important contributor of requirements.

—*Changing expectations*. In addition to the technical problem of tracking changed requirements, there is the social problem of informing stakeholders of the consequences of changes, as well as managing stakeholders' requests and their expectations of change. Research shows that user behavioral participation and psychological involvement positively influence user satisfaction of the development products [Barki and Hartwick 1989]. User participation is particularly effective during requirements development [Kim and Lee 1986; Leventhal 1995; Liou and Chen 1993–1994; Magal and Snead 1993; McKeen and Guimaraes 1997].

By managing conflict, organizations can manage these technical and social difficulties. RIM attempts to address such technical and social problems as part of a strategy to manage the conflicts that contribute to the essential difficulties of requirements engineering. It addresses many problems by supporting requirements traceability in a dynamic, multistakeholder environment [Gotel and Finkelstein 1995]. For example, by tracking the statements asserted by analysts and stakeholders as they enact a requirements dialog, developers can manage voluminous requirements and visualize the changes in requirements, the analyst team, or system stakeholders. Problems that are more social can also be addressed. For example, by tracking stakeholder statements, analysts can find trends (e.g., convergence or divergence) of expectations. RIM tools can even support the detection and resolution of multistakeholder requirements conflict (see Section 5).

1.3. Article Overview

This survey has three major themes. First, we define RIM and its history (Sections 2 and 3). Second, we examine basic research themes involving the RIM activities (Section 4). Third, we summarize research projects illustrative of RIM (Section 5). We conclude by showing that RIM has become a critical area of requirements engineering whose methods will lead to the development of systems with higher stakeholder satisfaction and fewer failures (Section 6).

2. ELEMENTS OF RIM

As its name implies, RIM is about requirements, interactions, and management. Requirements are descriptions of needs. Interactions can be understood by comparing requirements descriptions or analyses of their underlying implementations. Management focuses on the activities that uncover and resolve requirements interactions.

2.1. Requirements

Requirement has many definitions, each emphasizing an aspect of requirements engineering [Zave and Jackson 1997]. Central to any definition is a stakeholder need. For example, Davis [1993] stated that a requirement is "a user need or a necessary feature, function, or attribute of a system that can be sensed from a position external to that system" [Davis 1993]. From such a broad definition stem many specialized requirement types that analysts use to categorize requirements (for a more refined description, see Pohl [1997]):

—*System*. These requirements describe the type of system, such as hardware or software. There may even be development requirements concerning the development process (cost-effective, timely) or development aspects of the resulting product (reusable, maintainable, portable).

—*Functional and nonfunctional*. These requirements describe the form of service. Functional requirements describe a service relation between inputs and outputs. Nonfunctional requirements do not define a service, but instead describe attributes of the service provision, such as efficiency and reliability. Nonfunctional requirements are sometimes called *system qualities*.

—*Abstraction level*. Analysts describe requirements at different levels of abstraction. They can add new details and define them in more specialized subrequirements. By specializing or refining abstract requirements, or by generalizing detailed requirements, they define a requirements abstraction hierarchy.

—*Representation*. One requirement can have several representations. It may begin as an informal sketch, become a natural language sentence ("The system shall..."), and end as a more formal representation (temporal logic, SCR, Knowledge Acquisition in autOmated Specification of software (KAOS; see Section 5.4)).

2.2. Interactions

Requirements are part of a managed development activity. As such, their interactions may or may not be formally analyzed. In fact, many projects maintain informal requirements and only informally estimate their interactions. This leads to three broad characterizations of requirements interactions:

—*Perceived interaction*. Requirement descriptions seem to imply that satisfying one requirement will affect the satisfaction of another.

—*Logical interaction*. The requirements' logical descriptions imply a contradiction or conclusions that can be inferred only through their combined contributions.

—*Implementation interaction*. Requirements interact through the behaviors of the components that implement them. If the implementation is correct, these interactions will be the same as the logical interactions. If the implementation is incorrect or if environmental assumptions differ from modeled assumptions, implementation and logical interactions may differ.

Taking a more formal perspective can be helpful in understanding how requirements interact. Consider a set of requirements, R. If we take each requirement as a logical statement, then logical inconsistency occurs if False is a logical consequence of the set of requirements [Wing 1990]:

Logical Inconsistency:

$$\mathtt{R} \models \mathtt{False}$$

Logical inconsistency means that no model can satisfy the requirements. Practically, this means that no software behavior will satisfy the requirements.

Development seeks to create an implementation, *Impl*, that behaves correctly according to R:

Correct Implementation:

$$\mathtt{Impl} \models \mathtt{R}$$

This means that *Impl* exhibits the behaviors required in R.

Real systems are a bit more complicated. Behaviors of the system's environment impose their own constraints, denoted by E:

Correct Implementation within an Environment:

$$\mathtt{E}, \mathtt{Impl} \models \mathtt{R}$$

This means that *Impl*, within E, exhibits the behaviors required in R.

Figure 1 illustrates environment and implementation behaviors as sets [Jackson 1995] whose intersection represents the admissible behaviors of the implementation within its environment. Requirements define this intersection because they describe the behaviors of the implementation within an environment. We can view requirements interactions in the context of this conceptual illustration:

—*Requirements interaction*. Requirements $R1$ and $R2$ interact in environment E, if some conclusion, C, can be derived only when both are included in the requirements. This may be trivially satisfied where C is the conjunction of requirements.

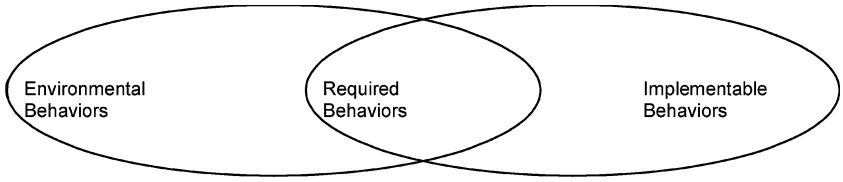

Fig. 1. Requirements as the boundary between environment behaviors and implementable behaviors.

Requirements interaction:
(1) $E, R1 \not\models C$
(2) $E, R2 \not\models C$
(3) $E, R1, R2 \models C$
(4) $C \not\models \mathtt{False}$

—*Negative interaction (conflict)*. A negative interaction is a kind of requirements interaction where False is derived from the conjunction of the requirements. In the preceding interaction definition, clause 4 becomes: $C \models \mathtt{False}$.

—*Positive interaction.* A positive interaction is a kind of requirements interaction where False is not derived from the conjunction of the requirements. In the preceding interaction definition, clause 4 is important, as C defines the effect of the interaction. (See Section 5.4 where van Lamsweerde's definitions of conflict and divergence account for inconsistencies among sets of requirements.)

—*Requirements implementation conflict.* Requirements $R1$ and $R2$ in environment E, interact through the behaviors of their respective implementations if each requirement can be individually satisfied by the component implementing it, whereas the two implementation components cannot be combined to satisfy both requirements together.

A requirements implementation conflict:
(1) $E, \mathtt{Impl1} \rightsquigarrow R1$
(2) $E, \mathtt{Impl2} \rightsquigarrow R2$
(3) $E, \mathtt{Impl1} \wedge \mathtt{Impl2} \not\rightsquigarrow R1 \wedge R2$

(In the preceding formula, the *satisfies* relation is denoted with $\rightsquigarrow$; *does not satisfy* is denoted with $\not\rightsquigarrow$ [Wing 1990].) It is possible that no implementation can satisfy both requirements:

All known requirement implementations conflict:
(1) $\exists\, \mathtt{Impl1} : E, \mathtt{Impl1} \rightsquigarrow R1$
(2) $\exists\, \mathtt{Impl2} : E, \mathtt{Impl2} \rightsquigarrow R2$
(3) $\neg\exists\, \mathtt{Impl3} : E, \mathtt{Impl3} \rightsquigarrow R1 \wedge R2$

Much requirements engineering literature addresses requirements descriptions and analyses, but less is said about how to describe environmental actions, laws, and norms, which can also be quite difficult. In some cases, the failure to consider environmental characteristics can result in disastrous system failures.

Flight systems are an example. In the air, braking is not allowed, but a safe landing obviously requires brakes. To ensure that pilots did not inadvertently engage the A320's braking system, the software required that the wheels detect the full weight of the airplane. However, when a Lufthansa pilot attempted to land in Warsaw on a wet, runway in high winds, the system did not detect the full weight of the plane on the wheels [Ladkin 1995a, 1995b], with the following results:

> [. . .] the spoilers, brakes and reverse thrust were disabled for up to 9 seconds after landing in a storm on a waterlogged runway, and the airplane ran off the end of the runway and into a conveniently placed earth bank, with resulting injuries and loss of life. [Ladkin 1995b]

Disastrous system failures, such as the Lufthansa A320, can arise from

undesirable interactions among requirements or between the required behaviors and the environment.

At some point, of course, the scope of the environmental model must have some boundary. Determining what about the environment to model is a development decision that depends on the available time and the budget. The assertions in E can determine if a requirements conflict exists. In fact, analysts can work backward. Given requirements $R1$ and $R2$, is there a condition, B, that could cause them to conflict? If so, is it likely that B will occur within the environment? B could be a rain-soaked runway, for example. Had such an obstacle been considered, the accident might not have happened. On the other hand, what is the likelihood of a rain-soaked runway? If it is low, its analysis can be postponed indefinitely. If it is high, analysis and modeling are more pressing. Analyst decisions regarding what to model can greatly influence the scope of the interactions that are uncovered.

2.3. Requirements for a Distributed Meeting Scheduler

The distributed meeting scheduler problem illustrates the challenges of analyzing requirements interactions. We chose this problem because the scheduler's requirements definition involve complex multi-stakeholder interactions that raise many issues like privacy, responsibility, and efficiency. We also chose it because its compact, yet rich, requirements document has been widely circulated [van Lamsweerde et al. 1993] and analyses have been published [Potts et al. 1994; van Lamsweerde et al. 1995; Robinson and Pawlowski 1997].

Van Lamsweerde et al. [1995, p. 197] stated the general requirements for a scheduler:

> The purpose of a *meeting scheduler* is to support the organization of meetings—that is, to determine, for each meeting request, a meeting *date* and *location* so that most of the intended participants will effectively participate. The meeting date and location should thus be as convenient as possible to all participants. Information about the meeting should also be made available as early as possible to all potential participants.

The remaining requirements of the four-page baseline description refine the roles of the meeting scheduler and participants. However, this introduction is sufficient to understand the examples that follow.

2.3.1. Requirements Definition. To show how requirements may interact, we present two requirements for information privacy and we describe them formally so that we can include examples of formal interaction analysis. The first is the `InitiatorKnowsConstraints` requirement [Robinson and Volkov 1997; van Lamsweerde et al. 1998]:

```
Requirement   InitiatorKnows
              Constraints
Mode          Achieve
InformalDef
          ''A meeting initiator shall
know the scheduling constraints of the
various participants invited to the
meeting within some deadline d (days)
after the meeting initiation.''
FormalDef
          ∀m : Meeting, p : Participant,
            i : Initiator
          Invited(p, m)
          ⇒ ◇≤dKnows(i, p.Constraints)
```

This requirement definition is in a variant of the KAOS language (see Sections 5 and 5.4), which allows for both informal and formal descriptions. The formal definition uses real-time temporal logic operators [Koymans 1992]. The $\Diamond$ means some time in the future. Other operators refer to the next state (o), the previous state ($\bullet$), some time in the past ($\blacklozenge$), always in the past ($\blacksquare$), and always in the future ($\Box$).

The second requirement for information privacy is, `InitiatorNever Knows-Constraints`:

```
Requirement   InitiatorNeverKnows
              Constraints
Mode          Avoid
```

```
InformalDef
        ''A meeting initiator shall
never know the scheduling constraints
of the various participants invited
to the meeting after the meeting
initiation.''
FormalDef
        ∀m: Meeting, p: Participant,
         i: Initiator
         Invited(p, m)
     ⇒ □¬knows(i, p. Constraints)
```

2.3.2. A Conflict and Its Resolution. The two requirements can conflict under certain circumstances, described in *boundary condition*, B [van Lamsweerde et al. 1998]. That is:

$$\texttt{InitiatorKnowsConstraints}, \texttt{InitiatorNeverKnowsConstraints}, B \models \texttt{False}$$

We can show a condition, B, that leads to a contradiction:

$$\Diamond\,(\exists\, \texttt{p}' : \texttt{Participant}, \texttt{m}' : \texttt{Meeting}\ \texttt{Invited}(\texttt{p}', \texttt{m}'))$$

The following shows that we can indeed derive a contradictory assertion from the conjunction of the two requirements and the boundary condition:

$$\Diamond\,(\exists\, \texttt{p}' : \texttt{Participant}, \texttt{m}' : \texttt{Meeting}\ \texttt{Invited}\,(\texttt{p}', \texttt{m}') \wedge \Diamond_{\leq \texttt{d}}\ \texttt{Knows}(\texttt{i}', \texttt{p}'.\texttt{Constraints}) \wedge \neg \Diamond\ \texttt{Knows}(\texttt{i}', \texttt{p}'.\texttt{Constraints}))$$

There are several ways to resolve this conflict [Robinson and Volkov 1996]. One is to rely on a scheduler program. Rather than provide scheduling constraints to the initiator, enable the scheduler to find a meeting time. Using this approach, only the scheduler program knows the constraints, while the initiator is simply notified of the meeting time. This approach is captured in the following requirement, which replaces

```
InitiatorKnowsConstraints:
Requirement  SchedulerKnows
             Constraints
Mode         Achieve
InformalDef
          ''A meeting scheduler shall
know the scheduling constraints of the
various participants invited to the
meeting within some deadline d (days)
after the meeting initiation.''
FormalDef
        ∀ m : Meeting, p : Participant,
          s : Scheduler
        Invited(p, m)
        ⇒ ◇≤d Knows(s, p.Constraints)
```

Relying on the scheduler to perform complex actions may be wishful thinking. That is, an implementation that satisfies the scheduler's required function may not exist. A step toward determining an implementation is to describe the scheduler's main function as follows:

```
Operation   DetermineSchedule
Input       MeetingRequest
Output      Meeting
PreCondition
           ¬Scheduled(m) ∧
(∃i: Initiator) Requesting(i, mr)
PostCondition
         Feasible(mr) ⇒ Scheduled(m)
∧ ¬Feasible(mr) ⇒ DeadEnd(m)
```

Still, the question remains: is there an implementation that either derives a meeting schedule or identifies infeasible meeting constraints (a dead end)? Moreover, this is simply one requirement. Analysts must still determine if all meeting scheduler requirements can be satisfied together. Additionally, the environment may cause problems, such as network delays in sending or receiving participant constraints, or reduced processing capacity on the (shared) computer. Therefore, while the initial requirements may be acceptable ($R \not\models \mathsf{False}$), analysts may need to elaborate them to ensure that they are satisfied within their environment (E, $Impl \models R$).

2.3.3. A Nonfunctional Conflict. Requirements may involve nonfunctional qualities of service, such as effort, cost, or usability. These can be analyzed in the same manner as the functional conflict between InitiatorKnowsConstraints and InitiatorNeverKnowsConstraints. Consider the following nonfunctional requirement (we have deliberately omitted formal definitions here):

```
Requirement SchedulerShallNot
            IncreaseParticipantEffort
Mode        Achieve
InformalDef
          ''A meeting scheduler for
a meeting shall not increase the
effort of invited participants."
```

SchedulerShallNotIncrease ParticipantEffort indicates that the scheduler shall not increase the effort of participants. This may be difficult to achieve because we assume that the scheduler is to determine a meeting time. To do so, it must know the current participant scheduling constraints. Thus, there is a functional assumption, InviteeRespondsWithUpdated Constraints, that captures the assumption that participants will reply to the scheduler's request with updated constraints:

```
Assumption   InviteeRespondsWith
             UpdatedConstraints
InformalDef
             ''A meeting participant
shall update his scheduling constr-
aints within some deadline d1 (days)
after the meeting request and then
reply to the request with his cons-
traints (within d2 days).''
```

A domain definition links the scheduler's behavior to the participant's behavior. The following RequestIncreasesEffort definition indicates that, by the request, the requesting agent has increased the effort of the receiving agent. This definition shows that the scheduler can increase the effort of a participant:

```
DomainDef  RequestIncreasesEffort
InformalDef
           "An agent that requests a
reply from a second agent imposes
an increased effort on the second
agent.''
```

It may be clear by now that there is a potential nonfunctional conflict, which can be summarized as

```
SchedulerShallNotIncreaseParticipant
Effort, InviteeRespondsWithUpdated
Constraints, RequestIncreasesEffort
⊨ False
```

The increased burden imposed by the scheduler conflicts with the decreased effort specified by the requirement SchedulerShallNotIncreaseParticipant Effort.

2.4. Interaction Features

Analysts can define requirements interactions through their features, such as basis, degree and direction, and likelihood.

2.4.1. Basis. The basis specifies the elements of the interaction. In the preceding example, a nonfunctional requirement, an environmental assumption, and a domain property together form the basis of the nonfunctional conflict. More specifically, the conflict basis is the minimal set of statements (environment and requirement) that imply a contradiction. (See Section 2.2.)

2.4.2. Degree and Direction. Some requirements imply a logical contradiction, and thus define a 100% conflict. However, analysts can specify requirements satisfaction as a range of values—1 to 100%, for example, see Liu and Yen [1996]. Given that requirements satisfaction may be partial, requirements interaction may be partially positive or partially negative.

Looking again at SchedulerShallNot IncreaseParticipantEffort, suppose we replace the statement that the scheduler shall "not increase the effort of invited participants" with the statement that the scheduler shall "minimize the effort of invited participants." Call this new requirement, Scheduler MinimizeParticipantEffort.

Given this formulation, analysts must determine if an increase in a participant's effort, caused by `InviteeResponds WithUpdatedConstraints`, conflicts with the scheduler's requirement to minimize participant effort. One conclusion is that such an increase "somewhat" conflicts with effort minimization, which suggests the use of a qualitative scale. Of course, multiple occurrences of an increase could lead to a "strongly" conflicts conclusion [Chung et al. 1995].

So far, we have focused mainly on negative interactions. Requirements may also interact to *reduce* conflict. Consider, for example, an automated reply mechanism for the participant in the scheduler problem. Given a request, another agent replies instead, thereby reducing the effort of the requested agent. The following domain definition describes this:

```
DomainDef  AutoReplyReducesEffort
InformalDef
          ''A proxy agent that replies to a request for another agent
decreases the effort for the other
agent."
```

Now, an analyst can conclude that `Auto ReplyReducesEffort` increases the satisfaction of `SchedulerMinimizeParticipant Effort`. Thus, the same requirements can have both positive and negative interactions. For example, if the scheduling system required both autoreply and participant reply, then the satisfaction of `SchedulerMinimizeParticipantEffort` would be indeterminate.

Others have modeled partial requirements satisfaction as follows: the degree of satisfaction, d, of a requirement, R, by the behaviors of implementation, *Impl*, is defined by the following ternary relation:

Partial Requirements Satisfaction:

```
PartSat(Imp, R, Degree), where
Imp        : set of implementations
R          : set of requirements
Degree     : [0, 100]
```

We can use $Impl \rightsquigarrow_{\mathrm{d}} R$ to indicate that *Impl* partially satisfies R to the degree, d. A simple and practical approach places scaled attributes on requirements [Gilb 1977, 1988]. Then, stakeholders associate values with the attributes, such as *Usability = 60 Availability = 20* (on a scale of 100). Analysts can use these metrics to identify unsatisfied requirements.

Some analysts have used fuzzy set theory to formalize requirements satisfaction into linguistic terms, such as high, or low. Fuzzy set theory maps ranges of satisfaction onto terms, as illustrated in the following definition of the PartSat fuzzy set [Liou and Chen 1993–1994; Liu and Yen 1996; Yen and Tiao 1997]:

```
FuzzySet_PartSat(Degree) = High, where
   Degree ∈ [75..100]
FuzzySet_PartSat(Degree) = Medium,
   where Degree ∈ [25..74]
FuzzySet_PartSat(Degree) = Low, where
   Degree ∈ [0..25]
```

Utility theory and fuzzy set theory provide techniques to aggregate requirements satisfaction across a variety of attributes, such as cost or reliability.

2.4.3. Conflict Likelihood. Often, requirements can conflict; but the likelihood of such a conflict may be acceptable. Regardless, characterizing the likelihood that an interaction will occur is always helpful.

In the scheduler problem, the degree of satisfaction for the requirement `SchedulerMinimizeParticipantEffort` depends on the environment. In an environment where a participant receives many requests from the scheduler, the satisfaction of `SchedulerMinimize ParticipantEffort` may be low ($Impl \rightsquigarrow_{\mathrm{Low}} R$). An analyst can model the number of requests, k, that a participant receives. Then, the likelihood that a participant receives many requests (P_k) determines the likelihood that the requirement `SchedulerMinimizeParticipantEffort` will be satisfied. If the analyst knows that $\mathrm{P}_k \approx 0$, or the consequences of `Scheduler MinimizeParticipantEffort` failing are acceptable, then it is likely that the requirements will be satisfactory within the specified environment.

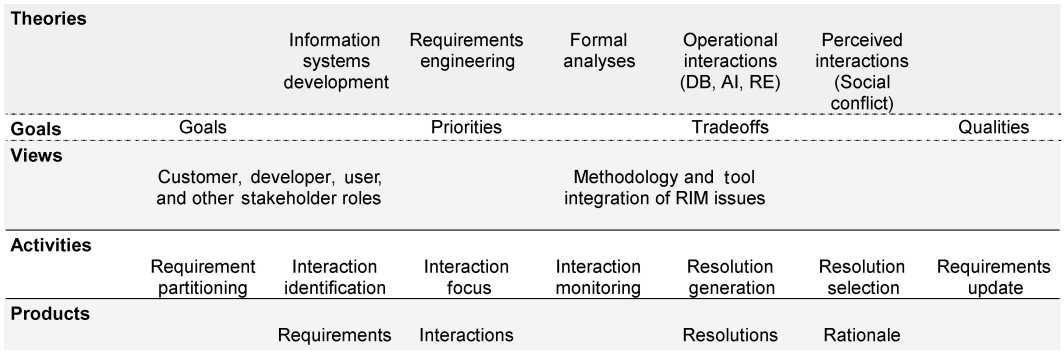

Fig. 2. A descriptive framework of RIM research.

2.5. Managing Interaction

The management element of RIM concerns the strategic application of activities to identify, analyze, monitor, document, communicate, and change requirements interaction. The activities may be applied within an ad hoc or a defined process, may involve the use of special tools and techniques, and may be conducted solely by analysts or by analysts and other stakeholders. In any case, the overall goals of these activities include the following:

—Detect and resolve requirements conflict (negative interactions).

—Increase system effectiveness by mutually reinforcing requirements (positive interaction)

—Increase involvement from a variety of stakeholders.

Satisfying these goals reduces overall system errors and costs and increases system effectiveness and stakeholder satisfaction.

As a discipline, RIM is new and evolving. It has five major dimensions:

—*Representation of requirements, interactions, resolutions, and other products*. Researchers are defining the set of terms, or ontology, that describe requirements and their interactions.

—*Activities for discovery, management, and disposition of interactions*. Researchers are defining techniques, some automated, that manage or aid in managing interactions. Often, this research aims to provide early life-cycle analysis rather than address interactions at system runtime.

—*Views of the activities and products*. RIM research is not conducted in isolation. Rather, it is being integrated into traditional software development tools and methods. Thus, different stakeholders may access different views of RIM analysis. Views include abstract agent descriptions found in *i** (see Section 5.3) to the tabular transition tables of SCR (see Section 5.6). Eventually, customers, developers, and users will be able to access requirements interactions from views tailored to their purpose.

—*Goals of stakeholders*. Researchers are defining goal ontologies and analyses to aid in requirements negotiations among stakeholder views and the strategic application of RIM.

—*Theoretic basis for representation, activities, and views*. Researchers are augmenting traditional requirements-engineering theories with theories from database, artificial intelligence, knowledge acquisition and representation, and social conflict and negotiation to establish a theoretical basis for the support and application of RIM.

Figure 2 proposes a classification of RIM research along these five dimensions. The *theories* provide a basis for developing new specialized techniques. Drawn from a variety of disciplines, they include concepts like database schema integration

that researchers can adapt to fit RIM. Other concepts include models, ontologies, and formal analyses for requirements engineering.

The *goals* dimension defines RIM goals and strategies. For example, a common social negotiation strategy suggests resolving simple conflicts first and difficult ones later [Pruitt 1981]. Such a strategy may also be appropriate for software development. However, defining and determining simple and difficult conflicts will be among the concerns in specializing the strategy. The goals dimension also defines goals for individual and collective stakeholder views.

The *views* dimension defines stakeholder interfaces to the activities and products that make up RIM's technological component. For example, an analyst typically has access to all the activities and their products during development, while a system user may have a more limited view. Similarly, the methodologies and tools used provide varied perspectives on RIM issues. For example, a tool may address the management of requirements interactions in support of a RIM-oriented development methodology, while another tool ignores RIM issues.

The *activities* dimension defines analyses and modifications for requirements interaction.

The *products* dimension includes intermediate and final results used during the activities.

3. AN HISTORICAL PERSPECTIVE

RIM has a narrow "systems" focus on interaction management, but it borrows from many theories and techniques from other disciplines.

3.1. Conceptual Evolution

Table I summarizes prominent concepts and their evolution into the emerging discipline of RIM. Because of space limitations, the references for each concept are representative, not exhaustive. We distinguish concepts by time and by the five categories described in Figure 2: theory, goals, views, products, and activities.

The top-most row of Table I summarizes some RIM theoretical developments. Many of these concepts, such as preference, conflict, negotiation, and resolution, derive from human negotiation [Pruitt 1981] and group decision-making [Janis and Mann 1979]. A general overarching tenet of RIM is analogous to that of group decision-making:

> Specifying stakeholder views on system requirements, followed by their negotiated integration, will result in systems that are both technically better, but are also more accepted by system stakeholders.

Goals and strategies make this tenet operational.

3.1.1. Development Goals Interaction. Researchers and practitioners recognized early on that specifying development goals is important. In a 1974 experiment, Weinberg and Schulman [1974] gave teams one of the following goals to satisfy: minimize effort, minimize lines of code, minimize memory use, maximize program clarity, and maximize clarity of program output. All but one team did best on their given goal. Since then, many researchers have specified a variety of software development goals, and their relationships [Barbacci et al. 1995, 1997; Boehm 1981; Chung et al. 1995; Kazman et al. 1998]. Most recently, the emphasis has been on creating models and tools to aid in the analysis of software development goal interactions. An example is WinWin [Boehm 1996], which we describe in Section 5.1.

Reasoning about requirements goals has evolved concurrently with the evolution of software development goals. Multiple Attribute Utility Theory (MAUT) [Raiffa 1968] and later Multiple Criteria Decision Making (MCDM) [Zeleny 1982] have provided general decision theoretic techniques that help analysts elicit criteria and trade them off during decision-making. An example is Oz [Robinson 1994], which we describe in Section 5.5. The more specialized decision technique

Table I. Evolution of RIM Concepts

	Before 1970	1970s	1980s	1990s
Theory		Codified negotiation techniques: "log rolling," condition restructuring [Pruitt 1981] Group decision-making [Heym and Osterle 1993]	Requirements Negotiation Behavior [Bendifallah and Scacchi 1989; Robinson 1990] Negotiation experts: case-based [Sycara 1991], rule-based [Werkman 1990b]	Domain-independent resolution generation [Robinson and Volkov 1996]
Goals	Management by Objectives [Drucker 1954]	Programming goals [Weinberg and Schulman 1974] Software qualities [Boehm et al. 1978] Software metrics [Gilb 1977]	Software development goal structure [Boehm 1981]	Software quality attributes [Barbacci et al. 1997] Software quality interaction experts [Boehm 1981]
	MAUT [Raiffa 1968]	MCDM programming [Zeleny 1982]	MCDM for requirements [Robinson 1994] QFD [Hauser 1988]	Software quality architecting [Kazman et al. 1998] QFD for requirements [Jacobs and Kethers 1994] Nonfunctional framework [Mylopoulos et al. 1992]
Views		Multiview specification [Mullery 1979]	Method Engineering [Kumar and Welke 1992] Parallel elaboration [Feather 1989]	Requirements Viewpoints [Nuseibeh et al. 1994]
			Process programming [Osterweil 1987]	Process compliance [Emmerich et al. 1997] Interaction monitoring [Fickas and Feather 1995]
Products			Requirements Modeling Language [Greenspan et al. 1994] Requirements traceability [Potts and Bruns 1988]	Goal oriented requirements [Dardenne et al. 1993] Agent-oriented requirements [Mylopoulos et al. 1997]
Activities		Goal-based design [Kant and Barstow 1981]	Goal-based requirements negotiation [Robinson 1989] Program slicing [Horwitz et al. 1989]	Goal regression for requirements [Robinson 1993, 1994; van Lamsweerde et al. 1998]
			Schema integration [Batini et al. 1986] Inconsistency dialog [Finkelstein and Fuks 1989]	Inconsistency reasoning [Hunter and Nuseibeh 1998] Inconsistency framework [Cugola et al. 1996]
			Multiagent planning [Georgeff 1984]	Agent negotiation [Sandholm and Lesser 1995]

of Quality Function Deployment [Hauser 1988] has been applied to requirements analysis (e.g., Jacobs and Kether [1994]).

3.1.2. Multiple System Views. Views, or views, on sets of related requirements have also evolved. Approaches, such as CORE [Mullery 1979], ETHICS [Mumford and Weir 1979], and later MultiView [Avison 1990] and Soft Systems [Checkland 1981], combined both social and technical development aspects in representing various system requirements views. Feather's [1989] parallel elaboration work described the algorithmic aspects of representing, comparing, and combining various system views, as did the ViewPoints project [Nuseibeh et al. 1994] (see Section 5.3), and a growing body of related research [Finkelstien 1996].

3.1.3. Requirements Modeling Languages. Requirements definition languages evolved to support reasoning about interactions among requirements views. For example, the Requirements Modeling Language (RML) [Greenspan et al. 1994] has given rise to languages that focus on agents [Mylopoulos et al. 1997] and goals [Dardenne et al. 1993]. (See KAOS the discussion of in Section 5.4). Such languages let analysts determine how the actions of external and system agents affect the satisfaction of system requirements.

3.1.4. Interaction Analysis. Many requirements-interaction-reasoning techniques have come from related fields. For example, goal regression, an Artificial Intelligence (AI) planning technique, can uncover certain requirements that are the root cause of a conflict [van Lamsweerde et al. 1998]. (See Section 5.5.) Similarly, database schema integration ideas [Batini et al. 1986] are helpful in combining requirements views [Spanoudakis and Finkelstein 1997]. Finally, from nonmonotonic reasoning have come logics and frameworks for reasoning about logical inconsistencies [Hunter and Nuseibeh 1998].

Table II. Disciplines Influencing Requirements Interaction Management

Software engineering
Requirements engineering
Formal specification
Concurrent engineering
Quality architecting
Feature interaction
Database view integration
Schema integration
Schema reengineering
Knowledge acquisition and representation
Knowledge integration
Information integration
Distributed artificial intelligence
Reasoning with inconsistency and incompleteness
Distributed problem solving, coordination, collaboration
Negotiation support systems
Coordination
Collaboration
Group issues: dominance, anonymity
Social conflict and negotiation
Negotiation theory and models
Negotiation strategies and tactics
Bargaining and arbitration
Political negotiation
Social economics
Individual decision-making
Cognitive dissonance theory
Utility theory

3.2. Influential Disciplines

Interaction management research, most of which aims to identify and manage negative interactions, spans a variety of disciplines. To characterize influences on RIM, we surveyed over 60 published works that RIM articles referenced. These referenced works described mainly theories, techniques, and tools for the management of conflicts. They included disciplines from the computer and information to cognitive and social sciences. From the articles, we identified seven distinct areas of influence, which Table II lists.

3.2.1. Software Development. Researchers in software development are addressing interaction management in a number of contexts, including the following:

—*Requirements inconsistency*. Detecting and resolving requirements inconsistency is a growing theme of requirements engineering, which we expand in this article. The Viewpoints

project (see Section 5.3), for example, provides a framework in which rules capture inconsistencies among and within views of a system requirements specification.

—*Formal specification*. Analysts have recognized the need for methods to detect and resolve inconsistencies in formal specifications [Lamsweerde 2000]. Now, formal techniques can detect inconsistencies [Spanoudakis and Finkelstein 1995], deduce in the presence of conflict [Hunter and Nuseibeh 1998], and manage conflict [van Lamsweerde et al. 1998] (see the discussion of KAOS in Section 5.4).

—*Concurrent engineering*. Detecting and resolving design differences among the designs of multifunctional and multidisciplinary teams is a concern of concurrent engineering [Kusiak 1993]. Quality function deployment (QFD) is commonly used to identify interactions among system requirements as well as among lower-level design or production requirements [Kusiak 1993]. Other methods, such as heuristic conflict classification and resolution identify and resolve undesirable design interactions [Klein 1991].

—*Feature interaction*. Detecting and resolving undesirable functional interaction is an established part of telephony software development. For example, there is a conflict between "Caller ID," which provides the receiver with the caller's number, and "Unlisted Number," which keeps the caller from revealing the originating number. An AI planning method that includes goal hierarchies generates a resolution in which the callee receives the caller's name, but not the caller's number [Velthuijsen 1993]. This planning method is but one of a number of methods for resolving feature interactions. A recent article surveys formal, informal, and experimental methods [Keck and Kühn 1998].

—*Quality architecting*. Analyzing how different system architectures affect tradeoffs among system qualities is a concern of software architects [Perry and Wolf. 1992]. A method, such as ATAM [Kazman et al. 1998], analyzes system qualities, such as performance, security, and reliability to determine if a system architecture can satisfy multiple interacting system qualities. If it cannot, the method helps select an architecture that satisfies the most qualities.

3.2.2. Database View Integration. Researchers in database development are addressing interactions in *schema consistency*. Traditionally, relational database designers start with multiple views of data and then combine those views into a global data schema. As part of the schema integration activity, they identify conflicts among the views [Batini et al. 1986], including differences in *name* or *structure*. Generally, defined methodologies have supported this activity [Batini et al. 1986], but tool support is growing [Francalanci and Fuggetta 1997; Jeusfeld and Johnen 1994; Johannesson and Jamil 1994; Ram and Ramesh 1995].

3.2.3. Knowledge Acquisition and Representation. Researchers in knowledge acquisition and representation are addressing interactions in a number of contexts, including the following:

—*Knowledge integration*. Knowledge bases, such as those in expert systems, should be consistent if they are to support deductive reasoning. To support this consistency goal, there must be some way to combine knowledge gained from multiple experts and make it consistent within the computerized knowledge base. Common knowledge-integration techniques include the use of meta-knowledge, set-theoretic analysis, consensus theory, repertory grid analysis, cluster analysis, and decision theory [Botten et al. 1989]. One tool, based on repertory grid and personal construct theory, aims to support the derivation of terminological consistency among experts [Shaw and Gaines 1988, 1989].

—*Information integration*. During its execution, a knowledge-base system may receive a variety of inconsistent inputs. To solve its overall task, the system must appropriately deal with these inconsistencies [Lander and Lesser 1989; Hearst 1998]. For example, various scheduling databases in a meeting scheduling system might reference the same person using slightly different names; an information integration agent can reconcile this discrepancy by recognizing naming differences [Sycara et al. 1996].

3.2.4. Distributed Artificial Intelligence. The analysis of interaction among distributed artificial intelligence agents is similar to the analysis of multiple viewpoint requirements. Each distributed AI agent represents a requirements viewpoint and the agent knowledge base represents the requirements viewpoint description. Thus, when distributed AI agents interact to complete shared tasks, their representation and reasoning is similar to that found in the integration of multiple requirements viewpoints.

Distributed AI research is addressing interactions in a number of contexts, including the following:

—*Distributed agent negotiation*. Distributed artificial intelligence has expanded the role of planning. Multiagent planning systems identify and resolve plan failures that occur among sets of loosely coordinated agents (e.g., Conry et al. [1991]; Durfee [1988]; Georgeff [1984]; Kraus and Wilkenfeld [1990]; von Martial 1992]). If the planners reach an inconsistent state, they may cooperatively negotiate to satisfy other plans [Conry et al. 1991]. To do so, they may use economic models to guide their decision-making so that they can efficiently manage their resources [Sandholm and Lesser 1995]. Again, the method of resolution is typically subgoal replanning; however, the subgoal failure and replanning is complicated without global information. Researchers in AI, and subsequently distributed AI, have defined negotiating agents, which has given rise to two complimentary research areas:

—*Negotiation analysis knowledge*. Case-based reasoning and rule-base programming codify and aid analyses to identify and resolve the conflicts that arise in a variety of domains, including labor negotiation [Sycara 1988], design integration [Klein 1991], and specification integration [Robinson and Volkov 1997].

—*Negotiation protocol knowledge*. Frameworks [Conry et al. 1988; Smith 1980] and communication protocols [Kraus and Wilkenfeld 1990; Lander and Lesser 1993; Mazer 1989; Oliver 1996; Sandholm and Lesser 1995] aid in coordinating the sequences of messages among distributed negotiating agents.

3.2.5. Negotiation Support Systems. Researchers in negotiation support systems (NSS) are addressing interactions with the aim of advising a human negotiator or supporting humans gathered around a negotiation table [Jelassi and Foroughi 1989, Lim and Benbasat 1992–1993]. Research ranges from developing an NSS "shell" aimed at supporting the construction of negotiation systems [Kersten and Szpakowicz 1994; Matwin et al. 1989] to specialized domain support such as airline buyout [Shakun 1991], product marketing [Rangaswamy et al. 1989], or electronic marketplace [Yen et al. 1996]. Some negotiation support systems have borrowed from AI reasoning features. Negotiation support systems often focus on human aspects of negotiation including the dominance and anonymity of participants.

3.2.6. Social Conflict and Negotiation. Theories and studies of conflict and negotiation among humans have influenced much of the research just described. This background in persuasion [Fisher and William 1991], negotiation [Pruitt 1981; Raiffa 1982], and decision-making [Janis and Mann 1979; Raiffa 1968] is the basis for many computerized models.

Researchers have adapted techniques from human negotiations to assist in

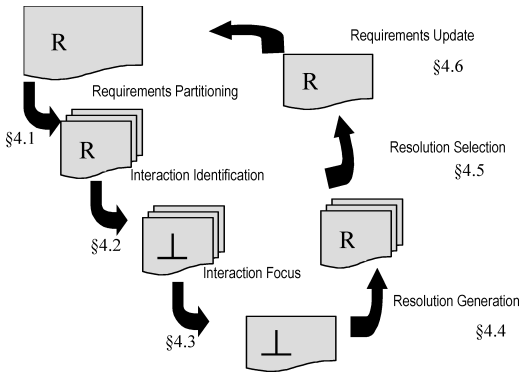

Fig. 3. An illustration of the activities that are managed as part of the requirements interaction management life-cycle. Each number indicates the section where an activity is introduced. The symbol, R, depicts a set of requirements, while the symbol, ⊥, depicts conflicting requirements.

detecting and resolving requirements conflicts. For example, case-based and rule-base methods [Robinson and Volkov 1996; van Lamsweerde et al. 1998] were generalized and formalized from the domain-specific techniques of the 1980s.

3.2.7. Individual Decision-Making. Individual decision-making has influenced the basic theories of interaction management research. Multiattribute utility theory suggests how an individual can trade off various interacting goal to maximize overall utility [Raiffa 1968]. Some decision models consider the dynamic aspects of this process. For example, as an individual learns of the tradeoffs among goals, he or she may reconsider the value, or weight, placed on individual goals [Zeleny 1982].

4. LIFE-CYCLE ACTIVITIES

Figure 3 illustrates RIM activities. We derived the figure from our model of automated negotiation, which we created from a survey of tools and theories [Robinson and Volkov 1998]. The description begins with unstructured requirements, which analysts may partition. Next, interaction identification may provide conflicts that analysts must address. Through interaction focusing, they consider only a subset of interactions at a time. Resolution generation provides alternative ways to resolve each conflict. Finally, resolution selection determines which resolutions will become change requests for the requirements document.

As Figure 3 shows, the activities in the RIM life-cycle fall into six groups. Each RIM activity has associated research issues, which we describe next.

4.1. Requirements Partitioning

Requirements partitioning seeks to focus interaction analysis on manageable requirements subsets. This is important, since analyzing all the interactions among *all* requirements can involve significant computation. For n requirements, there can be $n(n-1)/2$ binary conflicts; the space is even worse for n-ary conflicts. Partitioning seeks to divide this problem into a set of smaller problems. Requirements partitioning gives rise to the following issues:

Issue: *On what basis should a requirements document be partitioned into subsets that enhance analysis?*

Many computer science communities have considered problem partitioning based on goals, among them the distributed AI community [Gasser and Huhns 1989]. Partitioning based on stakeholder views is a commonly used natural partitioning based on the originating source of the stated requirements [Checkland 1981; Gotel and Finkelstein 1995]. Other partitions may be based on root requirements [Robinson and Pawlowski 1998], requirements issues [Yakemovic and Conklin 1990], nonfunctional software attributes [Boehm 1996], requirements consistency [Easterbrook 1993], or scenarios [Potts et al. 1994]. In composite system design, Feather [1989] has used the cross-product of functional partitioning and agent responsibility to partition requirements. This approach is continued in the deficiency driven design method, described in Section 5.5.

Issue: *How can an analyst allocate requirements to partitions?*

Table III. Types of Requirements Interactions

Type	Description	Exemplar
Positive interaction	Increasing the satisfaction of **R**1 increases the satisfaction of **R**2.	Some, +, ++, [Chung et al. 1995] Influence + [Gustas 1995]
Negative interaction	Increasing the satisfaction of **R**1 decreases the satisfaction of **R**2.	Hurts, −, −−, [Chung et al. 1995] Contradictory Influence - [Gustas 1995]
Unspecified interaction	Changing the satisfaction of **R**1 has an unspecified effect on the satisfaction of **R**2.	Impacts on interdependency
No interaction	Increasing the satisfaction of **R**1 has no effect on the satisfaction of **R**2.	Neutral

If each partition has some characterization, database and keyword search technology can partition requirements. For example, commercial tools can apply database technology to select subsets of requirements based on requirement attributes. Difficulties arise when the requirements are not attributed a priori with necessary characteristics. In such cases, partitioning can be based on the keywords in each requirement. However, the presence of a keyword does not necessarily indicate that the key characteristic is in the requirement. Some researchers have overcome the limitations of keyword retrieval by using concept-based information retrieval [Chen 1992; Chen et al. 1993].

Sometimes requirements subsets constructed with different terminology can further confound partitioning—for example, when different people develop the requirements. In such cases, statistical measures of usage can help generate mappings among terms [Shaw and Gaines 1988].

Issue: *Given a requirements partitioning, how can analysis of the partitions be ordered to enhance analysis?*

Strategies for the ordered analysis of requirements partitions are rare. A general software life-cycle strategy, such as the spiral model, considers the riskiest partitions first [Boehm 1988]. Specialized approaches consider cost-benefit analysis [Karlsson and Ryan 1997; Cornford et al. 2000] or contention [Robinson and Pawlowski 1998]. However, this research area seems largely unexplored.

4.2. Interaction Identification

In Section 2.2 we introduced the term, *requirements interaction*, and its two subtypes, *negative interaction* and *positive interaction*. Classifying interactions and detecting conflicts have their own sets of issues.

4.2.1. Classifying Interactions. A variety of fields have contributed to a growing classification of interaction types, particularly AI and requirements engineering.

4.2.1.1. AI-Based Classifications. AI planning concepts have introduced types such as goal/subgoal decomposition. Goal conflict itself is explained in terms of conditions, or resources, of operators that attempt to achieve a goal; for example, goals may conflict because operators that satisfy goals individually have interfering preconditions when achieved simultaneously. Similarly, two requirements may conflict because they mutually deplete available system resources. Many such planning terms have also been applied to characterize interactions among requirements [Potts 1994; Robinson 1989; Robinson and Volkov 1996].

4.2.1.2. Requirements-Engineering-Based Classifications. Table III summarizes the most general types of interactions found in the requirements-engineering-related literature: positive, negative, and unspecified.

A more refined analysis of the literature reveals the basis of most interactions. Table IV summarizes these refined types, which include interactions over structure, resources, task, causality, and time.

Table IV. Basis of Requirements Interactions

Type	Description	Exemplar
Structure	**R**1 is similar to **R**2.	Duplicate, alternative [Chung et al. 1999]
Resource	**R**1 and **R**2 depend on the same resource.	Resource utilization/contention [Yu and Mylopoulos 1993]
Task	**R**1 describes a task required for **R**2.	Subtask, means/ends, operationalization [Dardenne et al. 1993; Yu and Mylopoulos 1993]
Causality	**R**1 describes a consequence of **R**2.	Results in Moffett [2000]
Time	**R**1 has a temporal relation to **R**2.	Coincident state, simultaneity constraint, pre/post time relation [Malone and Crowston 1994]

Some researchers have found that a domain model based simply on positive and negative types is a practical solution [Boehm 1996; Chung et al. 1995; Dardenne et al. 1993; Ramesh and Dhar 1992; Yakemovic and Conklin 1990; Robinson and Pawlowski 1998]. Others have extended such work to incorporate fuzzy logic concerning the degree of conflict [Yen and Tiao 1997]. In more recent work, fault-trees were the basis for classifying interactions types. A fault tree, such as that concerning human-computer-interaction requirements [Maiden et al. 1997], is viewed as an a priori enumeration of common negative interactions between the software and its environment. In a similar fashion, a classification of common interactions among functional [Klein 2000] and nonfunctional software development attributes can serve as the basis for annotating how requirements conflict [Boehm 1996]. Some researchers have considered interactions other than conflict, such as the cost/benefit of requirements [Karlsson and Ryan 1997; Cornford et al. 2000].

4.2.2. Conflict Detection Methods. Zave and Jackson [1993, p. 404] suggested that practical consistency checking will largely be language dependent:

> In theory, the consistency of a multiparadigm specification could be investigated within predicate logic, after translating all partial specifications into that form. In practice, this is obviously infeasible. The logical formulas resulting from the translation are large and incomprehensible, and the complexity of a real specification in that form would be far beyond the capacity of existing automated tools.
>
> We believe that most practical consistency checking must be formulated at the same conceptual level as the specification languages used, and that algorithms for consistency checking will be specialized for particular languages and styles of decomposition. . . .
>
> It is already common practice for researchers to work on analyzing and verifying specifications within particular application areas or written in particular languages, and they are beginning to work on verifying specifications of popular system architectures. We are not proposing any change to this practice except the use of a small set of complementary languages instead of one language, which should make the overall goal easier to achieve.

The authors show how features common to different specification languages can guide the translation of a multiparadigm specification into a logic for consistency checking [Zave and Jackson 1996]; others have done similar work [Ainsworth et al. 1996; Niskier et al. 1989; Nuseibeh et al. 1994]. Delugach [1992, 1996] translated specifications into conceptual graphs for detection. Rather than rely on a common semantic domain, Fiadeiro and Maibaum [1995] used category theory to detect inconsistencies among multiparadigm specifications. Still, many current interaction-detection techniques rely on certain features of the specification language.

Table V summarizes five categories of methods for evaluating requirements for possible interactions. All the methods address the following issue:

Issue: *What kinds of analyses can be applied to requirements to uncover requirements interactions?*

4.2.2.1. Classification-Based. A general classification can help in identifying

Table V. Interaction Detection Methods

Method	Description	Example
Classification-based	Requirements interactions are found and classified by comparing requirements against an a priori model of requirements interactions.	WinWin [Boehm 1996], NFR [Mylopoulos et al. 1992], CDE [Klein 1991]
Patterns-based	Requirements are compared with detection pattern conditions. An interaction is found when there is a match. Resolution patterns derive resolutions in a similar manner.	KAOS [van Lamsweerde and Letier 2000]
AI planning	Requirements interactions and resolutions are found through planning. Requirements are represented as goals, while operations are represented as planning operators.	DDR (Oz) [Robinson 1993], KAOS [van Lamsweerde and Letier 2000]
Scenario analysis	Requirements interactions are demonstrated by simulating a sequence of events that represents a narrow aspect of a system's required behavior.	SCR [Heninger 1980], CREWS-SAVRE [Maiden, 1998; Sutcliffe et al. 1998]
Formal methods	Requirements interactions are found by algorithmic verification (SPIN, SMV), deductive verification (PVS, HOL), and language-specific verification (SCR, RSML).	SPIN [Holzmann 1997], SMV [McMillan 1992], PVS [Crow et al. 1995; Owre et al. 1995], HOL [Gordon and Melham 1993], SCR [Heninger 1980], RSML [Leveson et al. 1994]
Runtime Monitoring	Requirements interactions are found by monitoring a system execution for certain events that indicate violations of requirements specifications.	FLEA [Feather 1997; Feather et al. 1998]

requirements interactions. Classification captures commonly occurring interactions among requirements and environment features [Fox et al. 1996; Gruninger and Fox 1995; Olsen et al. 1994; Storey et al. 1997]. A hierarchy of such *binary interactions* typically defines the classification. Most work on nonfunctional interactions uses such binary interactions to indicate interactions [Boehm 1996; Chung et al. 1995; Dardenne et al. 1993; Ramesh and Dhar 1992; Robinson and Pawlowski 1998].

Classification-based interaction detection proceeds in three phases: (1) classify requirements according to the interaction classification, (2) instantiate the associated general interactions from the classification, and (3) infer requirements interactions from the instantiated domain interactions. Detection can be automated [Spanoudakis and Constantopoulos 1996], and the analysis of even five simple interaction types yields significant benefits [Robinson and Pawlowski 1998].

The following definition relates information accuracy and the effort required to provide such information:

```
DomainDef  AccuracyIncreasesEffort
InformalDef
          ''Accurate information in-
creases effort of information provi-
ders.''
```

Now, consider two nonfunctional requirements $R1$ and $R2$ belonging to the `Accuracy` and `Effort` categories, respectively. The interaction model may indicate a negative interaction, which implies that

$R1$ and $R2$ might interact negatively. This is summarized in the following rule:

```
AnalysisDef  DetectNonFunctional
             NegativeInteraction
InformalDef
        ''If two requirements belong
to nonfunctional categories that the
interaction model says are negati-
vely interacting then the two requi-
rements might be interacting negati-
vely.''
```

An analogous rule holds for positive interactions.

As an example, consider the `Scheduler ShallNotIncreaseParticipantEffort` requirement of Section 2.3.3, along with the following `ParticipantAccurate Constraints` requirement.

```
Requirement  ParticipantAccurate
             Constraints
Mode         Achieve
InformalDef
             ''A meeting particpant's
constraints shall be accurate.''
```

`SchedulerShallNotIncrease ParticipantEffort` references `Effort` while `Participant AccurateConstraints` references `Accuracy`. `DetectNon FunctionalNegativeInteraction` instantiates `AccuracyIncreasesEffort` along with the two requirements to infer that requiring accurate participant constraints will increase participant effort.

WinWin is probably the best known requirements tool that uses a classification of binary interactions to notify stakeholders of new requirements interactions [Boehm 1996; Egyed and Boehm 1996]. (See Section 5.)

4.2.2.2. Patterns-Based. An interaction classification of binary interactions can lead to wrong inferences about interactions. Accurate information (`Participant AccurateConstraints`) will not always increase participant scheduling effort (`SchedulerShallNotIncreaseParticipant Effort`), for example. Computerized selection and scheduling of a meeting room can *decrease* the effort of meeting participants as well as increase schedule information accuracy, which implies an *increase* in participant effort as they maintain their on-line information. Thus, a classification of binary interactions is simple to construct but, if not precise, can lead to wrong conclusions.

A more precise analysis is possible with interaction patterns. Such patterns have pre- and postconditions that constrain their use and the conclusions they draw. Consider the following simple interaction pattern:

> An activity performed by a software agent that produces information used by a human agent decreases the human agent's effort.

This pattern provides more conditions of use than binary patterns of the form *X increases Y*. Consequently, few wrong inferences will be drawn. Of course, a multiplicity of patterns leads to the same problem of inconsistency that occurred with binary interaction classification. In this case, however, because the patterns are precise, the problem is not as severe.

The KAOS project defines formal patterns [van Lamsweerde et al. 1998]. The Critic specification tool [Fickas and Nagarajan 1988] and Klein's concurrent requirements analysis tool (CDE; [Klein [1991]) and his Conflict Detection Section of the MIT Process Handbook [Klein 2000] use interaction patterns to detect conflicts.

4.2.2.3. AI Planning. It is possible to automatically compare requirements and classify their interaction type. Given operational requirements, program slicing techniques [Horwitz et al. 1989; Yang et al. 1992] can highlight semantic differences in versions of a common root specification [Heimdahl and Whalen 1997]. For requirements represented as nonoperational systems goals, planning techniques aid in deriving a plan for the conjunction of the requirements set:

(1) If the planner finds a plan in a given operator set, then requirements *can* be achieved simultaneously.
(2) If it fails to find a plan, analysts can use goal regression to find the reason for the requirements conflict [Fickas and

Anderson 1989; Robinson 1993, 1994; van Lamsweerde and Letier 2000].

Finally, a set of requirements can succeed and fail in the same environment. Analysts can check for this by planning for the conjunction of some requirements with the negation of others; if such a plan succeeds, then the requirements can fail in the way the plan demonstrates [Fickas and Anderson 1989]. Of course, the operations and other environmental resources must first be formalized for some part of the system environment [Fickas and Anderson 1989]. Such a planning approach to goal interaction is also suitable for scenario analysis: requirements become plan goals and the plan operators become the scenario actions. Analysts generate scenarios by considering a variety of plan failures, such as precondition failure via resource depletion [Fickas and Anderson 1989; Maiden et al. 1997]. Distributed AI research also uses such interaction detection [Gasser and Huhns 1989; Velthuijsen 1993; von Martial 1992].

4.2.2.4. Scenario Analysis. Van Lamsweerde and Willemet [1998, p. 1089] defined a scenario as follows:

> A temporal sequence of interaction events among different agents in the restricted context of achieving some implicit purpose(s). . . . A scenario captures just one particular, fragmentary instance of behavior of a system.

Because scenarios represent system execution fragments, analysts can evaluate their outcome relative to requirements. *Positive* scenarios satisfy requirements, while *negative* scenarios violate them. A scenario that is both positive and negative still violates certain requirements, and thus shows a negative interaction.

In scenario analysis, the analyst selects a subset of requirements to be analyzed and then identifies a particular scenario to determine if the selected requirements can be satisfied. If the analyst finds no such scenario, he or she has in effect discovered a negative interaction. For temporal logic requirements and state-based scenarios, model checking tools can automate scenario analysis [McMillan 1992; Holzmann 1997]. Alternatively, a knowledge-based approach can suggest scenarios that are likely to generate requirements interactions [Maiden 1998]. Finally, scenario-based advice can be distilled into checklists or fault-trees as a way to uncover interactions manually [Leveson 1995].

4.2.2.5. Formal Methods. Formal methods are used for a variety of purposes in analyzing specifications, according to van Lamsweerde and Jetier 2000] (cf [Heitmeyer and Mandrioli [1996]):

—to confirm that an operational specification satisfies more abstract specifications, or to generate behavioral counterexamples if not, through algorithmic model checking techniques [Queille and Sifakis 1982; Clarke and Emerson 1986; Holzman 1991, 1997; McMillan 1993; Atlee 1993; Manna and Group 1996; Heitmeyer et al. 1998b; Clarke et al. 1999];

—to generate counterexamples to claims about a declarative specification [Jackson and Damon 1996];

—to generate concrete scenarios illustrating desired or undesired features about the specification [Fickas and Helm 1992; Hall 1995, 1998] or, conversely, to infer the specification inductively from such scenarios [van Lamsweerde and Willemet 1988];

—to produce animations of the specification in order to check its adequacy [Hekmatpour and Ince 1988; Harel et al. 1990; Dubois et al. 1993; Douglas and Kemmerer 1994; Heitmeyer et al. 1996; Thompson et al. 1999];

—to check specific forms of specification consistency/completeness efficiently [Heimdahl and Leveson 1996; Heitmeyer et al. 1996];

—to generate high-level exceptions and conflict preconditions that may make the specification unsatisfiable [van Lamsweerde et al. 1998; van Lamsweerde and Letier 2000];

—to generate higher-level specifications such as invariants or conditions for

liveness [Lamsweerde and Sintzoff 1979; Bensalem et al. 1996; Park et al. 1998; Jeffords and Heitmeyer 1998];

—to drive refinements of the specification and generate proof obligations [Morgan 1990; Abrial 1996; Darimont and van Lamsweerde 1996];

—to generate test cases and oracles from the specification [Bernot et al. 1991; Richardson et al. 1992; Roong-Ko and Frankl 1994; Weyuker et al. 1994; Mandrioli et al. 1995];

—to support formal reuse of components through specification matching [Katz et al. 1987; Reubenstein and Waters 1991; Massonet and van Lamsweerde 1997; Zaremski and Wing 1997].

Many of the preceding methods support of conflict detection, as well as other RIM activities. The SCR tool kit, for example, finds missing, ambiguous, and erroneous requirements [Bharadwaj and Heitmeyer 1997; Heitmeyer et al. 1996; Schneider et al. 1998]; see Section 5.6. Using the KAOS approach, an analyst can uncover conflicts among requirements or between requirements and the environment; see Section 5.4.

4.2.2.6. Runtime Monitoring. Runtime monitoring tracks the system's runtime behavior for any deviations from the requirements specification. Requirements monitoring is useful when verifying system properties is too difficult, when resolving noncritical conflicts during specification may be too costly or lead to unnecessary restrictions, when assumptions made about the environment are evolving, or when specific dispositions must be deployed for specific use modes. During requirements definition, developers integrate certain assumptions, which analysts then monitor at runtime. Should the assumptions fail, monitoring invokes a predefined procedure, such as to notify the designer. Monitoring differs from exception handling in three ways: (1) it considers the combined behavior of events that occur in multiple threads or processes; (2) it links runtime behavior with the actual design-time requirements; (3) it provides enough information for developers to reconfigure the software or software components at runtime.

Fickas and Feather [1995] proposed requirements monitoring that tracks the achievement of requirements at runtime as part of an architecture that lets developers reconfigure component software dynamically. Feather's working system, called FLEA (for Formal Language for Expressing Assumptions Language Description), lets developers monitor events defined in a requirements monitoring language [Feather et al. 1997, 1998]. Constructs in the language are mapped to triggers in a specialized database. As FLEA records interesting events in the database, the database triggers provide alerts when requirements fail.

Fickas and Feather [1995] illustrated runtime requirements monitoring for a software license server. When the license server fails to satisfy its requirements (e.g., a user shall be granted a license in 90% of their requests) because of a change in the system environment, the system notifies an administrator. As a follow-on, Robinson's [2002] ReqMon demonstrated the use of assertion checking to link runtime monitors to requirements.

Expectation agents monitor the system's actual use [Girgensohn et al. 1994]. Developers define software user expectations, such as, "validate the customer address before configuring the customer's services." Agents then monitor the system's use and when it does not match the defined expectations, agents perform the following actions: notify developers of the discrepancy, provide users with an explanation based on developers' rationale, and/or solicit a response to or comment about the expectation [Girgensohn et al. 1994]. Thus, in one sense, expectation agents monitor the satisfaction of developer expectations.

4.3. Interaction Focus

Issue: *Given a number of requirements interactions, how can the interactions be partitioned to enhance the analysis?*

Some requirements interactions depend on other requirements interactions. For example, the resolution of one conflict may introduce new conflicts into the requirements set; conversely, one resolution may remove multiple conflicts.

Efficient resolution focuses on key interactions. A necessary first step in generating a resolution is to partition or order interactions for consideration. In general, any such ordering should aim first to decrease overall conflict and minimize rework. Robinson has ordered requirements by their degree of *contentiousness*—the percentage of conflicting interactions that a particular requirement has among all requirements [Robinson and Pawlowski 1998]. Resolution generation then focuses on resolving conflicts among requirements with the greatest total contention. This strategy monotonically decreases the overall conflict in a small requirements document [Robinson and Pawlowski 1998], but it considers only conflict dependencies, not factors such as the importance of the requirements. Less contentious requirements may be a priority because the system *must* achieve them *exactly* as stated (no negotiation). In that case, the focus would be on conflicts that involve requirements with the greatest total importance. However, with an importance focus, resolution generation considers relatively few requirements. Thus, resolutions may be myopic. For example, in a limited-resource environment, assigning all resources to important requirements solves the important conflicts, but can introduce new conflicts, as some requirements cannot be satisfied. Thus, analysts must weigh a requirement's importance in the context of its implications for other requirements. Research into cost-value tradeoffs of requirements aims to assist this activity [Karlsson and Ryan 1997; Cornford et al. 2000].

Techniques for partitioning a large set of requirements are also effective in partitioning requirements interactions. (See Section 4.1.) Here, the focus is on all interactions in a particular partition: stakeholder, scenario, requirements subsumption hierarchy, and so on. In fact, decision science suggests that individuals can maximize their own benefit by first understanding and specifying their own preferences before negotiating with others [Zeleny 1982]. This suggests using stakeholder partitioning to resolve partition conflicts *within* a partition before attempting to resolve them *among* partitions. Moreover, the negotiation literature suggests that, in social contexts, resolving the simplest conflicts first builds trust among the negotiating participants [Pruitt 1981].

4.4. Resolution Generation

Issue: *Given requirements interactions, how can resolutions be generated?*

Conflict resolution can be characterized as a multiple goal-planning problem: given goal sets $G1$ and $G2$ held by agents $A1$ and $A2$, respectively, the resolution activity attempts to find a combined goal set similar to $\{G1, G2\}$ that the system can achieve without conflict. Resolution is commonly characterized as a tuple—agents, goals, environment—in which multiple agents seek to achieve goals within an environment, and the environment specifies available operators, resources, and other domain constraints.

Several approaches have successfully automated conflict resolution [Robinson and Volkov 1998]. Table VI summarizes six categories of conflict-resolution methods described in the requirements-engineering literature. We derived the six categories from approximately 29 methods, 11 of which we identified as unique. (Table IX summarizes projects that use these methods.)

The compromise method in Table VI is an example of a *value-oriented* approach to conflict resolution. The method searches for alternative goals (or goal values) to find nonconflicting substitute goals (or goal values). If the substitute goals are ordered, lexicographical ordering [Zeleny 1982] (to find less desirable

Table VI. Conflict Resolution Methods

Method	Description
Relaxation: generalization, value-range extension	Conflicting requirements are relaxed to expand the range of mutually satisfactory options beyond what the original requirements specify. Generalization involves replacing the conflicting concept with a more general concept. Value-range extension changes the range of values acceptable to the stakeholders.
Refinement specialization	Conflicting requirements are decomposed into specialized requirements, some of which can be satisfied.
Compromise	Given a conflict over a value within a domain of values, compromise finds another substitute value from that domain.
Restructuring: reenforcement, replanning	Restructuring methods attempt to change the conflict context; they alter assumptions and related requirements in addition to the conflicting requirements. Restructuring attempts to reduce constraining interactions and allow a wider range of resolution options. Reenforcement is a restructuring that ensures a precondition is satisfied. Replanning is the selection of an alternative set of requirements in order to achieve a subordinate requirement.
Other: postponement, abandonment	Conflict resolution can be postponed. In complex interactions, many conflicts and requirements are interrelated. By postponing a conflict and resolving other conflicts, the postponed conflict might cease to exist. Alternatively, conflicting requirements can be abandoned.

goals) characterizes resolution generation as a constraint relaxation problem. [Conry et al. 1988; Werkman 1990a, 1990b]. If the goals are arranged in an AND/OR hierarchy, replanning can generate alternative goal values [Adler et al. 1989; Velthuijsen 1993; von Martial 1992].

The other methods in Table VI are examples of a *structure-oriented* approach, which considers new operators and resources, as well as resource sharing. Also called "lateral" "out of the box thinking" [Fisher and William 1991], the structure-oriented approach is considered more likely to lead to an optimal resolution, than the value-oriented approach [Kersten and Szpakowicz 1994; Pruitt 1981], because it provides the freedom to redefine both goals and the environment. Thus, it is not surprising that many knowledge-based agents use some form of problem restructuring to generate resolutions [Kersten and Szpakowicz 1994; Klein 1991; Matwin et al. 1989; Sycara 1988, 1991]. Although many distributed AI projects use the term *negotiation* to describe their work, conflict resolution typically involves some form of compromise, goal relaxation, or goal dropping [AAAI 1994; Kannapan and Marskek 1993; Kwa 1988; Mostow and Voigt 1987; Sathi et al. 1986; Werkman 1990b]. In general, the focus of these projects is not on defining new resolution-generation techniques, but on incorporating resolution into a distributed AI architecture.

One version of the structure-oriented approach matches new conflicts with a domain-dependent case base that contains associations of previous conflicts and their resolutions [Klein 1991; Sycara 1988, 1991]. Analysts can then derive new resolutions from the resolutions of similar previous cases. Another implementation approach encodes structure-oriented resolution knowledge into a domain-dependent rule-based system [Kersten and Szpakowicz 1994; Matwin et al. 1989].

Researchers can generalize such restructuring transformations using basic negotiation principles. A theory-based, rather than domain-based, approach overcomes problems attributed to the narrow expertise of expert systems [Buchanan and Shortliffe 1984]. Negotiation theory-based domain-independent transformations apply across application domains and still apply when faced with unforeseen circumstances [Robinson 1997; Robinson and Volkov 1996; van Lamsweerde et al. 1998]. (For example, see Section 5.4.)

4.5. Resolution Selection

Issue: *Given requirements resolutions, how does the analyst select the "best" resolutions?*

Decision science theories suggest how to select the best alternative from a set of alternatives. The classic approach is based on *utility*—the benefit derived from an alternative [Raiffa 1968]. An overall utility decomposes into multiple criteria, to form a multiple-criteria utility function [Zeleny 1982]. In decision science, a criterion is the same as a requirement attribute. Thus, if requirements are given scaled nonfunctional attributes, then stakeholders can use those attributes to specify that they seek to maximize, minimize, or reach specific attribute values. Robinson [1993, 1994, 1997] has demonstrated the value of using stakeholder multiple-criteria utility functions to derive a preference ordering among requirements or conflict resolutions. Jacobs and Kethers [1994] also demonstrated a specialization of this general approach, based on the House of Quality methodology. Robinson [1993, 1994, 1997] has also suggested the value of Zeleny's [1982] Interactive Decision Evolution Aid (IDEA)—an interactive decision procedure that provides feedback on criteria tradeoffs among decision alternatives. The analyst does not explicitly specify tradeoffs among criteria but rather seeks to improve values of the current solution along specific criteria. The analyst invokes resolution generation to create new alternatives in the search for a good resolution. As this activity progresses, the analyst will focus on a succeedingly narrower solution set, thus subjectively determining the optimal solution [Festinger 1964; Janis and Mann 1979]. The result is in effect a settlement of tradeoffs among stakeholder positions [Robinson 1994; Zeleny 1982].

Issue: *Can a strategy integrate resolution selection into resolution generation?*

Given that analysts know which solutions are preferable, resolution generation would be more efficient if it could incorporate that knowledge and thus limit the search to the most promising resolutions. An approach that incorporates selection information would weed out all but resolutions with the preferred characteristics using some kind of matching For example, a case-based approach finds resolutions by matching conflict contexts [Klein 1991; Sycara 1988, 1991]. Neural networks provide another means of matching [Oliver 1996].

The drawback of a match-based approach is the difficulty of explaining what led to a specific resolution. Alternatively, the transformations in a transformation-based approach can provide textual explanations of the reasoning [Neches et al. 1985] and formal analyses of the transformations applied (e.g., refinement [Darimont and van Lamsweerde 1996]). Moreover, given selection preferences, an approach can incorporate those preferences into preconditions of transformation to make generation more efficient [Mostow and Voigt 1987].

4.6. Methodological Issues

The RIM life-cycle activities described in the previous five subsections raise methodological issues about when and in what context to use them.

Issue: *When should the life-cycle activities take place?*

Methodological guidance as to when and why analysts should engage in requirements management activities is rare. Traditional approaches, such as the classic software life-cycle, suggest checking and resolving interactions after any substantial change to a document—especially, after each life-cycle phase [Boehm 1998]. Some methodologies explicitly seek independent, and possibly inconsistent, partitions [Checkland 1981; Mullery 1979; Nuseibeh et al. 1994; Robinson and Volkov 1996; Schuler and Namioka 1993]. Nuseibeh [1996] summarized ways in which various software methodologies address conflicts in descriptions, including ignoring, circumventing, removing, and ameliorating them. He went on to suggest metrics that analysts should track as part

of inconsistency management, including the likelihood of failures due to unresolved conflict, and dependent decisions that the status of a conflict might affect. Unintrusive "reminders" could aid in tracking conflict status. Unfortunately, not much work considers when to apply conflict detection and resolution if conflicts are unresolved for some time.

Issue: *How can automated support for interaction management be provided for multiple analysts?*

Many projects address the management of requirements interactions for multiple analysts. Chen and Nunamaker have proposed a collaborative CASE environment, tailoring GroupSystems decision-room software, to facilitate requirements development [Chen 1991]. Using C-CASE, the software tracks and develops requirements consensus. Potts et al. [1994] have defined the Inquiry Cycle Model of development to instill some order into analyst dialogs about requirements interactions—specifically, interactions that arise as part of scenario analysis. The model develops requirements from analyst discussions and categorizes them as questions, answers, and assumptions. By tracking these types of dialog elements (and their refinements), the model maintains dialog while keeping inconsistency, ambiguity, and incompleteness in check through specific development operations and requirements analysis (e.g., scenario analysis).

The ViewPoints project has stimulated substantial research into the management of multiple requirements representations [Easterbrook 1994; Finkelstien 1996; Finkelstein et al. 1994a; Kotonya and Sommerville 1996; Mullery 1979; Sommerville and Sawyer 1997]. At its core is the representation of multiple development documents, which may be in different languages (dataflow diagrams, Petri nets, and so on), as well as different stakeholder views (Manager, Employee, and so on).

WinWin also provides groupware support for tracking team requirements development, including conflict detection and resolution [Boehm 1996; Egyed and Boehm 1996]. In addition to issue tracking, the tool aids conflict characterization with its hierarchy of common requirements conflict criteria.

Still other collaborative CASE efforts use meta-models to aid analysis across stakeholder views [Nissen et al. 1996; Ramesh and Dhar 1992; Hahn et al. 1991].

In addition to direct support for analyzing the requirements themselves, many projects indirectly support requirements analysis by giving analysts tools to aid dialogs about requirement analysis. Basic problems of collaborative CASE include information control, sharing, and monitoring [Vessey and Sravanapudi 1995]. Collaboration problems include how to support task, team, and group analysis [Vessey and Sravanapudi 1995]. Collaborative tools like electronic whiteboards and videoconferencing can capture the dialog surrounding analysis. Tools can link such rationale to the requirements dialog in the way that similar tools link source documents to specific requirements [Christel et al. 1993]. Analysts can adapt these collaborative tools to aid conflict resolution and capture rationale for selected resolutions.

Issue: *How can the status of the activities be monitored: through development and through system operation?*

Most collaborative CASE tools support the tracking of document annotations, updating a document's annotated status as it passes from one activity to another. Fewer tools support the explicit specification, achievement, and tracking of methodology goals. For example, consider the goal, "All requirements must have a defined user priority":

$$\forall\, \mathtt{R} \in \mathtt{Requirement}, \exists\, \mathtt{P} \in \mathtt{UserPriority} \bullet (\mathtt{HasPriority}\ \mathtt{R}\ \mathtt{P})$$

(This requirement supports standard PSS05, which specifies that under incremental development, all requirements will have a user-defined priority [Mazza et al. 1994].) It is desirable to support

Table VII. RIM Support in Seven Projects

Activity	WinWin	NFR	ViewPoints	KAOS	DDRA	SCR	M Telos
Requirements partitioning	A+	A	M	A	N	A	A
Interaction identification	A+	A	A+	A+	A+	A	A+
Interaction focus	A	A	M	N	A	N	N
Resolution generation	A	M	A	A+	A+	N	N
Resolution selection	A	A+	N	N	A+	N	N

Key: M = some manual support, A = some computer automation, A+ = computer automation specifically designed to solve the problem, N = no support described.

analysts in their specification, achievement and tracking of such methodology goals.

Workflow and process modeling may provide some solutions for managing requirements development [Sheth 1996]. Some approaches, for example, generate a work environment from a hierarchical multiagent process specification [Miller et al. 1997]. Others have incorporated such process models into CASE tools [Mi and Scacchi 1992], but these tools generally enforce processes constraints.

Process enforcement solutions ignore Osterweil and Sutton's [1996] observation about task sequencing:

> Experience in studying actual processes, and in attempting to define them, has convinced us that much of the sequencing of tasks in processes consists of reactions to contingencies, both foreseen and unexpected. [Osterweil and Sutton 1996, p. 159]

Requirements development is fraught with contingencies, as requirements and their dependencies are discovered as part of the requirements development methodology. Thus, requirements interactions management tools tend to downplay process enforcement and support the expression and monitoring of methodology goals [Emmerich et al. 1997; Robinson and Pawlowski 1999].

Emmerich et al. [1997] have illustrated how a tool can monitor the violation of methodology goals as part of a process-compliance-checking technique. Their work builds on Fickas and Feather's [1995] requirements-monitoring concept. (See Section 4.2.2.6.) Similarly, DealScribe can monitor a changing requirements document according to process goals. By applying consistency rules, it automatically detects inconsistencies, sends appropriate notifications, and even applies resolution methods [Robinson and Pawlowski 1999].

5. ILLUSTRATIVE PROJECTS

We have identified seven projects that illustrate some aspect of RIM. Although each project has a different research goal, each provides representations of requirements interactions, reasoning about interactions and conflict removal, and computer support for analysis. Together, the seven projects give a sample of the support possible for RIM.

Tables VII though IX provide an overview of the conflict detection and resolution support for the seven projects. Table VII provides an overview according to the activities of Figure 3. Table VIII overviews the interaction support provided by the projects. (The interaction types are defined in Table XI in Section 5.4.) Finally, Table IX provides an overview of the conflict resolution support according to the methods of Table VI. In all three tables, we subjectively assigned a value as a means to overview the level of automated support. We inferred the values from the project literature, because the literature does not directly indicate each project's support for all issues.

We now summarize each project in terms of the RIM activities and products.

5.1. Win Win

The WinWin project supports collaboration among a wide set of system stakeholders. Too often, software development solutions satisfy only a subset of all stakeholders, the "winners." Table X illustrates the distribution of winners and losers

Table VIII. Support for Conflict Detection Methods in Seven Projects

Interaction type	WinWin	NFR	ViewPoints	KAOS	DDRA	SCR	M Telos
Process-level deviation A deviation in development enactment from the defined development process.	N	N	A+	N	N	N	N
Instance-level deviation A class instance of the implementation violates requirements.	N	N	N	N	N	N	N
Terminology clash A single real-world concept is given different terms in the requirements.	N	N	A	N	N	N	N
Designation clash A term in the requirements designates multiple real-world concepts.	N	N	A+	N	N	A	N
Structure clash A term in the requirements is represented with multiple structures.	M	N	A	N	N	N	A
Conflict A set of requirements are logically inconsistent.	A	A	A+	A+	A+	A+	A+
Divergence A set of requirements can be shown to be logically inconsistent when a certain sequence of environmental and system events can occur.	M	A	N	A+	A+	A+	N
Competition A kind of divergence where particular instances of a requirement class cause a divergence.	M	A	N	A+	A+	A+	N
Obstruction A kind of divergence where a requirement fails when a certain sequence of environmental and system events can occur.	M	A	N	A+	A+	A+	N

Key: M = some manual support, A = some computer automation, A+ = computer automation specifically designed to solve the problem, N = no support described. KAOS automation is described rather than implemented.

Table IX. Support for Conflict Resolution Methods in Seven Projects

Method	WinWin	NFR	ViewPoints	KAOS	DDRA	SCR	M Telos
Relaxation: *generalization, value-range extension*	M	N	N	A+	A+	N	N
Refinement: *specialization*	M	N	N	A+	A+	N	N
Compromise	M	N	N	A+	A+	N	N
Restructuring: *reenforcement, replanning*	M	N	N	A+	A+	N	N
Other: *postponement, abandonment*	M	N	N	A+	A+	N	N

Key: M = some manual support, A = some computer automation, A+ = computer automation specifically designed to solve the problem, N = no support described. KAOS automation is described rather than implemented.

Table X. Frequent Software Development Win-Lose Patterns

Proposed solution	"Winning" stakeholders	"Losing" Stakeholders
Quick, cheap, sloppy product	Developer, Customer	User
Lots of features ("bells & whistles")	Developer, User	Customer
Driving too hard a bargain	Customer, User	Developer

Source: Table reproduced from Boehm et al. [1994].

for some typical software development solutions.

To have a winning outcome for *all* stakeholders, WinWin offers software support for multistakeholder requirements analysis and integrates such analysis into the larger software development life-cycle.

As Figure 4 shows, the development of winning stakeholder requirements is an activity. The activity model combines the risk reduction strategy of the spiral model [Boehm 1988] with the negotiation-oriented philosophy of Theory-W [Boehm 1989]. In each cycle, developers do

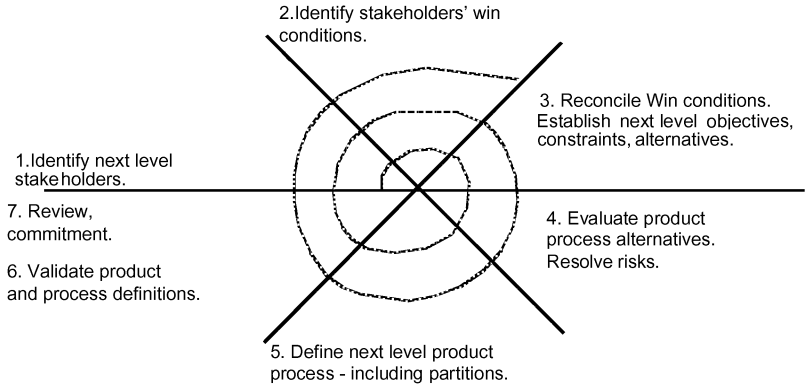

Fig. 4. The WinWin spiral activity model.

the following:

—Identify stakeholders.

—Identify the requirements of each stakeholder, called *win-conditions*.

—Identify requirements interactions, called *Conflict/Risk/Uncertainty Specifications* (*CRU's*) and capture their resolution as *Points of Agreement* (*POA's*).

—Elaborate the product and activity descriptions according to the new requirements. They also consider alternative means of satisfying the new collaborative requirements and select solutions with an eye toward risk reduction.

—Plan, validate, and review the next cycle.

The WinWin spiral activity model has three major milestones: Life-Cycle Objectives (LCO), Life-Cycle Architecture (LCA), and Initial Operational Capability (IOC). Requirements are among the six attributes that characterize each milestone. Stakeholders must commit to milestones between project inception, elaboration, and construction [Boehm et al. 1978].

5.1.1. Activities. The WinWin tool supports collaborative requirements analysis among stakeholders. In the context of RIM life-cycle activities (see Figure 3), this support includes the following:

—*Partitioning*. WinWin lets analysts partition requirements according to their associated attributes. These include project-defined attribute types, such as those listed in Figure 5 (from Boehm and Egyed [1998]). Each attribute type is also linked to stakeholder roles, interattribute relationships, and strategies for reducing conflict. The attribute types enable certain kinds of analysis, including the following:

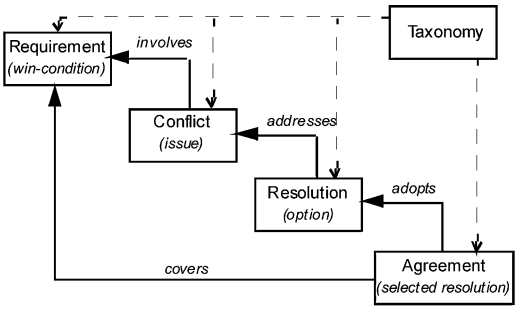

Fig. 5. WinWin artifacts. (Adapted from Boehm and Egyed [1998].)

—*Attribute importance.* Stakeholders vary in the importance they place on attribute types. QARCC, a component of WinWin, predefines an association of stakeholder roles to attribute types. For example, the User stakeholder role cares about Usability and Performance, while the Developer stakeholder role cares more about Cost and Schedule [Boehm 1996].

—*Attribute conflicts.* Analysts may know that certain attribute types conflict with others. In QARCC, each

1 Media operations 1.1 Query/Search/Browse 1.2 Access Control 1.3 Audio/Video Operation 1.4 Update/Input 1.5 Others	2 Interface 2.1 COTS (SIRSI, etc.) 2.2 Database (File Access) 2.3 User/Admin. Interface 2.4 Others	3 Administration 3.1 User Management 3.2 Usage Monitoring 3.3 Others	4 Quality 4.1 Response Time 4.2 Reliability 4.3 Security 4.4 Usability 4.5 Interoperability 4.6 Workload 4.7 Cost 4.8 Schedule 4.9 Others

Fig. 6. A WinWin project-specific taxonomy of attribute types.

attribute type has an associated set of supportive and detracting attribute types, which QARCC uses to identifying potential requirements conflict.

—*Attribute conflict reduction*. Analysts may also know that certain attribute type conflicts can be reduced through a set of general strategies. In QARCC, each attribute type has an associated set of textual activity and product strategies that stakeholders may consider when confronted with a conflict.

—*Identification*. When the analyst enters a requirement (win-condition) into QARCC, it generates a list of potentially conflicting requirements. To generate the list, QARCC first retrieves the attribute types associated with the new requirement, then retrieves the associated set of potentially conflicting attribute types, and finally finds the set of other requirements with those types [Boehm 1996]. For each such conflict, QARCC sends a Conflict Advisor Note message to stakeholders who have indicated concern about the attribute type.

—*Focus*. WinWin artifacts, such as requirements and resolutions, can be sorted by artifact attributes, such as owner, status, priority, revision date, etc.

—*Resolution*. When a potential conflict is identified, QARCC presents users with a list of predefined (text) strategies that may apply to the given attribute type conflict. A user may use this information to define a resolution Option.

—*Selection*. The user selects a resolution Option from the set of Options that all users have defined.

5.1.2. Products. Major requirements artifacts in WinWin include requirements, conflicts, resolutions, and agreements. Figure 5 illustrates the relationship among them. WinWin defines requirement terms in a project-specific taxonomy (Figure 6). However, QARCC does have an attribute-type taxonomy that includes an a priori model of requirements interactions and resolution strategies. This model enables QARCC to provide lists of potential requirements conflict and suggestions on possible ways to resolve them.

5.1.3. Case-Study Results. Since 1995, the WinWin project members have published articles describing case studies of software analysis using WinWin. Several results of those case-studies are noteworthy for RIM:

—*Between 40% and 60% of requirements involved conflicts.* In a 2-year comparison of projects involving 37 student teams, a significant number of requirements raised conflict [Egyed and Boehm 1998].

—*Most conflicts were simple to resolve.* This result seemed to depend on the complexity of the project, stakeholders' domain experience, and development resource constraints. Moreover, the time to create even a single resolution could be significant. Nevertheless, in 37 WinWin projects, between 45% and 69% of conflicts required only one resolution

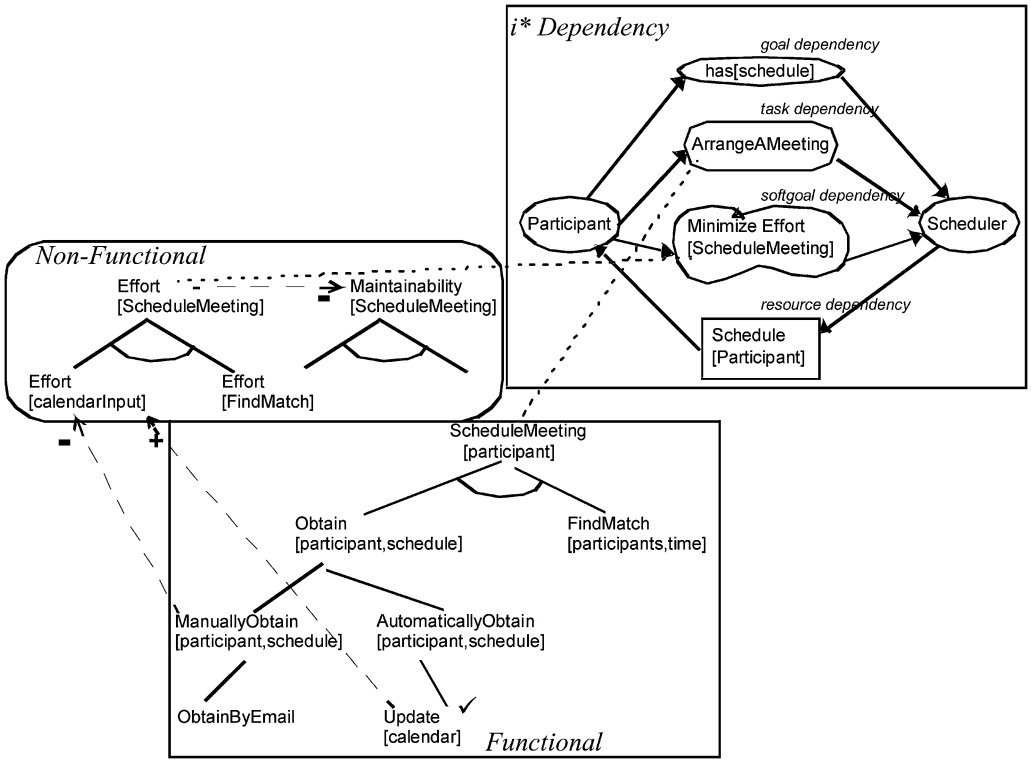

Fig. 7. An illustration of functional and nonfunctional and requirements, and the i^* model.

option before stakeholders reached an agreement [Egyed and Boehm 1998].

—*Developers contributed the most to identifying and resolving conflicts*. The stakeholder roles of User, Customer, and Developer contributed in varying ways to the project. Users and Customers contributed more to requirements identification, while Developers contributed more to conflict and resolution identification [Egyed and Boehm 1998].

—*System quality (nonfunctional requirements) involved the greatest conflict.* The next most controversial attribute type, operations, had nearly half as many conflicts as system qualities [Boehm and Egyed 1998].

5.2. Nonfunctional Requirements

Since 1992, the requirements project members from the University of Toronto have published articles describing the modeling and analysis of nonfunctional requirements and agent-oriented requirements (i^*) [Mylopoulos et al. 1992]. Although the literature typically presents these two topics separately, we present them together here because they are similar in the context of RIM.

Formal modeling of high-level business requirements has been the mainstay of this work. With the Requirements Modeling Language (RML) [Greenspan et al. 1994] as a precursor, the Toronto group has developed and formalized requirements semantics, methods for elaborating requirements, and relationships among requirements, including the linkage of agent-oriented system requirements to high-level business requirements.

5.2.1. Products. Figure 7 illustrates functional and nonfunctional requirements in relation to an i^* model of agent-dependencies, where i^* denotes the distributed intentionality among

the agents. The i^* model "views an organization as a network of *intentional* dependencies among actors in a social environment" [Yu and Mylopoulos 1993, p. 484]. The dependencies, in turn, provide a context in which to understand how functional and nonfunctional requirements interact in relation to organizational intentions.

Functional requirements, such as `ScheduleMeeting`, are represented as an AND/OR hierarchy [Mylopoulos et al. 1999]. Thus, Figure 7 illustrates that to achieve `ScheduleMeeting`, two subrequirements must be achieved: obtain participant schedules, and find a match among the schedules.

Nonfunctional requirements shown in the middle part of Figure 7 are also represented as an AND/OR hierarchy. Because nonfunctional requirements describe the qualities of functional requirements, Figure 7 parameterizes them by topic—a class that a requirement may reference. (Figure 7 shows requirement names instead of the more indirect topic names to reinforce the relationship between nonfunctional and functional requirements.)

The satisfaction of nonfunctional requirements can be aggregated. The overall satisfaction of a nonfunctional requirement is determined by combining the satisfaction of its subrequirements. For example, the `Effort` of `ScheduleMeeting` is determined by the `Effort` to obtain `calenderInput` and the `Effort` to `FindMatching` schedules.

Interactions between nonfunctional and functional requirements are diagramed—either supporting (+) or detracting (−). For example, `Update[calendar]`, which automates the task of providing updates to the system and thus reduces participant effort, supports the nonfunctional requirement of (minimizing) `Effort`. These interactions can be useful in determining the satisfaction of a nonfunctional requirement. For example, `Update[calendar]` has been satisfied (indicated with a check-mark "√" in the bottom center of the functional requirement box). If all AND subrequirements have similarly been satisfied, the overall requirement is satisfied. Of course, the satisfaction of other requirements may interfere. As Figure 7 shows, `ManuallyObtain[participant,schedule]` detracts from the satisfaction of `Effort`. However, the figure does not show that `ManuallyObtain[participant, schedule]` (or its descendents) has been satisfied, so `Effort[calenderInput]` is indeed satisfied.

In general, determining the satisfaction of a nonfunctional requirement can be quite difficult. Some functional requirements can detract from it; others can support it. Moreover, the satisfaction of one nonfunctional requirement can detract from another. In Figure 7, the negative link from `Effort` to `Maintainability` means that satisfying `Effort` (via minimizing) will detract from the satisfaction of `Maintainability`.

Despite such complexity, a qualitative label propagation algorithm has been developed that can determine if a nonfunctional requirement has been satisfied, denied, or undetermined [Mylopoulos et al. 1992]. Applying the algorithm to Figure 7, the satisfaction of `Update[calendar]` is propagated to `AutomaticallyObtain [participant,schedule]`, and then to `Obtain[participant,schedule]`. However, the algorithm labels `Schedule Meeting[participant]` as undetermined because `FindMatch[participants,time]` is undetermined and it is an AND subgoal of `ScheduleMeeting[participant]`.

Research has produced catalogs of nonfunctional requirements interactions [Chung et al. 1994, 1995] that describe how the satisfaction of one nonfunctional requirement can detract from the satisfaction of another. (See Section 4.2.) Although these catalogs are project specific, nonfunctional requirement interactions are arranged in a hierarchy of abstractions parameterized by topic. Thus, the more abstract nonfunctional requirement descriptions are reusable across projects.

In the i^* model at the top of Figure 7 [Mylopoulos et al. 1997], Participant and Scheduler are the actors, denoted by ovals. Although the model shows the intentional

relationships of organizational actors, it is accepted practice to show the intention of the automated Scheduler.

The model shows four dependencies:

—*Goal dependency*. The `Participant` depends on the `Scheduler` to have the goal `has[schedule]`. It is unspecified how the `Scheduler` may satisfy this goal.

—*Task dependency*. The `Participant` depends on the `Scheduler` to carry out the task `ArrangeAMeeting`. `Arrange AMeeting` could be a task that implements the functional requirement `ScheduleMeeting[participant]` that satisfies the goal `has[schedule]`; however, such intermodel relationships have not yet been defined.

—*Soft-goal dependency*. A soft goal has "no clear-cut definition and/or criteria as to whether it is satisfied or not" [Chung et al. 1999, p. 4]. The `Participant` depends on the `Scheduler` to perform some task that satisfies the soft goal `Effort[ScheduleMeeting]`. Like the goal dependency, the soft-goal dependency does not indicate which task the `Scheduler` should do to satisfy `Effort[ScheduleMeeting]`.

—*Resource dependency*. The `Scheduler` depends on the `Participant` to make his or her schedule available; this resource is denoted as `Schedule[Participant]`.

5.2.2. Activities. The University of Toronto projects have resulted in a number of prototypes. Collectively they provide considerable support to RIM activities:

—*Partitioning*. The framework allows for the partitioning of requirements into functional and nonfunctional type hierarchies parameterized by *topic* classes. Moreover, requirements can be stored in an object-oriented database [Jarke et al. 1995], so other attributes can be used to partition requirements. Finally, the interrelationship of the requirements and the i^* model can be used to partition requirements.

—*Identification*. When a requirement is entered into a CASE tool and associated with existing nonfunctional requirements, it can be determined, via the a priori supports and detracts links, which other requirements are directly affected. Label propagation from selected requirements can determine the cumulative effect of requirements on nonfunctional requirements satisfaction.

—*Focus*. Requirements can be sorted by their attributes and their associated interactions.

—*Resolution*. When an analyst identifies a negative interaction, she or he can generate alternative resolutions and add them to the requirements OR nodes.

—*Selection*. An analyst can select a requirement as a resolution by marking it as selected ("$\surd$"). A CASE tool supports the recording and analysis of claims, for or against, requirement decomposition. Such argumentation can be used to select a requirement [Chung et al. 1996].

5.3. Viewpoints

Since 1992, ViewPoints project members from Imperial College have published articles describing multiviewpoint modeling and analysis of systems [Finkelstein et al. 1992]. Their ViewPoints framework provides a means to partition requirements and analyze relationships between partitions [Nuseibeh et al. 1994]. (The work of Sommerville et al. [1997] described a closely related project called PreView [Kotonya and Sommerville 1996; Spacappietra and Parent 1994]. "Unlike the viewpoint model proposed by Finkelstein et al., our model is geared to elicitation and is not primarily intended for requirements validation" [Sommerville et al. 1997, p. 3].)

ViewPoints research addresses the integration of the heterogeneous representations that are normally part of requirements development. In particular, the research addresses [Nuseibeh et al. 1994] the following:

(1) the integration of the methods used to specify system requirements,

(2) the integration of the tools that support these methods, and
(3) the integration of the multiple specification fragments produced by applying these methods and tools.

ViewPoints researchers have developed formalisms and tools that aid in reasoning about inconsistencies within and among viewpoints.

5.3.1. Products. The ViewPoints framework supports multiple views of requirements. A view typically captures only a portion of the overall system description (i.e., a *partial specification*). Moreover, views of a system can vary in scope, representation, stakeholder ownership, or other dimensions. Easterbrook and Nuseibeh [1996, p. 37] summarized ViewPoints, as follows:

> ViewPoints are loosely coupled, locally managed, distributable objects which encapsulate partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of development.
>
> Each ViewPoint has the following slots:

- a representation style, the scheme and notation by which the ViewPoint expresses what it can see;
- a domain, which defines the area of concern addressed by the ViewPoint;
- a specification, the statements expressed in the ViewPoint's style describing the domain;
- a work plan, which comprises the set of actions by which the specification can be built, and a process model [Finkelstein et al. 1994b] to guide application of these actions;
- a work record, which contains an annotated history of actions performed on the ViewPoint.

Analysts can determine relationships among ViewPoints by the applying consistency rules. These rules detect inconsistencies among or within ViewPoints. ViewPoints can be represented in different languages: for example, data flow diagrams [Nuseibeh et al. 1994] or state transition diagrams [Finkelstein et al. 1994b]. Each rule has the following form (see Easterbrook [1994]):

$\forall$ VP_S: VP_{source}(t, d)
  logicalQuantifier VP_D :
    $VP_{Destination}$(t', d')
VP_S $\Re$ VP_D
Where, logicalQuantifier is either
  $\forall$ or $\exists$

This rule pattern stipulates the following:

—for any given source viewpoint of type t and domain d, there is a corresponding destination pattern of type t' and domain d' linked to it through relation $\Re$, or
—every pair of source viewpoints of type t and domain d and destination pattern of type t' and domain d' are linked through relation $\Re$.

As an example, the following interviewpoint rule expresses that "Process names must be unique across all DFD's" [Easterbrook 1994, p. 215]:

$\forall$ VP_S: VP_{source}(DFD, d_a)
  $\forall$ VP_D: $VP_{Destination}$(DFD, d_a)
    VP_S.ProcessName $\neq$
    VP_D.ProcessName

(In the above rule, d_a indicates that the destination can be any domain.) It is important to note that such rules apply from one viewpoint with respect to other viewpoints; internal viewpoint consistency is checked using other rules. Thus, the above check fails if the source viewpoint, VP_S, has a DFD process name that is also found in another viewpoint (of any domain). Complex rules, and interrule relationships, can be expressed within the ViewPoints framework.

5.3.2. Activities. As part of the ViewPoints project, computerized support has been specified; some support is provided in the software prototype, called the *Viewer* [Nuseibeh and Finkelstein 1992]. Collectively, the specified ViewPoints tools could provide considerable support to RIM activities:

—*Partitioning*. The ViewPoints framework allows for the partitioning of requirements into any subsets an analyst chooses. The framework itself does not provide categories; rather, it provides

the views into which one can place requirements.

—*Identification*. Once requirements are represented in ViewPoints, an analyst can apply the consistency rules to determine inconsistencies between ViewPoints.

—*Focus*. Inconsistencies correspond to rule violations. Thus, an inconsistency can be labeled by the corresponding rule being violated. It has not been demonstrated how the framework could prioritize the further analysis or the resolution process. However, interrule relationships can specify dependencies for the ordered application of consistency rules [Easterbrook 1994].

—*Resolution*. An analyst may apply a resolution rule that is associated with a violated consistency rule [Easterbrook 1994]. Alternatives can be considered in the scope of a hierarchy of consistent ViewPoints [Easterbrook 1993]; however, in general, resolution alternatives are not explicitly represented.

—*Selection*. Resolutions are not explicitly represented in the ViewPoints framework; thus resolution selection is not directly supported.

5.4. KAOS

Since 1991, the KAOS project members from the Université catholique de Louvain have published papers on modeling and analysis of system requirements [van Lamsweerde et al. 1991]; KAOS denotes *Knowledge Acquisition in autOmated Specification* of software. The project is broad in its scope, and includes meta-modeling, specification methodology, learning, and reuse.

5.4.1. Products. The KAOS language provides two basic levels of descriptions: "an outer semantic net layer for *declaring* a concept, its attributes and its various links to other concepts; an inner formal assertion layer for *formally defining* the concept" [Darimont and van Lamsweerde 1996, p. 181]. At the semantic net layer, KAOS provides an ontology of classes (cf. Dardenne et al. [1993] and van Lamsweerde et al. [1998]):

—*Object*. An object is a thing whose instances may evolve from state to state. (An object is similar to a UML class.) An object can be specialized to be an entity, relationship, or event if the object is autonomous, subordinate, or instantaneous, respectively.

—*Operation*. An operation is an input-output relation over objects; it defines state transitions.

—*Agent*. An agent is an autonomous object that can perform the operations assigned to it.

—*Goal*. A goal represents an objective that a system should meet. Goals can be refined into an AND/OR directed acyclic graph.

—*Requisite, requirement, assumption*. A requisite is a goal that can be formulated in terms of states controllable by a single agent. In other words, a requisite is a goal that can be assigned to an agent with the expectation that the agent can satisfy the goal through performing operations. A requirement is a requisite that has been assigned to a software agent. An assumption is a requisite that has been assigned to an environmental agent.

—*Scenario*. Typically, scenarios show how goals can be achieved. A scenario is a composition of operation applications. A composition of scenario operations must satisfy the pre- and postconditions of operations. Additionally, objects can have associated domain invariants that must be satisfied.

5.4.2. Activities. As part of the KAOS project, computerized support has been specified; some support is provided in the GRAIL software prototype [Darimont et al. 1998]. Collectively, the specified KAOS tools could provide considerable support to RIM activities:

—*Partitioning*. All KAOS objects, including requirements, are stored in an object-oriented database. Thus, requirements and their associated categories,

patterns, objects, attributes, and agents can be used to partition requirements [Darimont et al. 1998].

—*Identification*. KAOS defines requirements interaction types, as summarized in Table XI. Conflict is defined as a logical inconsistency among assertions. Divergence, perhaps the most interesting interaction type, is an inconsistency between goals and the environment. Consider a requirements divergence in resource management requirements [van Lamsweerde et al. 1998]. A user requirement might state, "If a user is using a resource, then she will continue to use the resource until it is no longer needed":

```
Requirement  UserContinuesUseOf
             Resource
Mode         Achieve
InformalDef
         ''If a user is using a
resource, then she will continue
to use the resource until it is no
longer needed.''
FormalDef
     ∀ u:User, r:Resource
     Using(u,r) ⇒o [Needs(u,r) →
       Using(u, r)]
```

In contrast, the library staff might state a requirement that "If a user is using a resource, then she will no longer do so after some d days":

```
Requirement  UserDiscontinuesUseOf
             Resource
Mode         Achieve
InformalDef
    ''If a user is using a reso-
urce, then she will no longer do
so after some d days.''
FormalDef
        ∀ u:User, r:Resource
        Using(u,r) ⇒◇≤dUsing(u,r)
```

Although the two requirements are not logically inconsistent, a problem does arise when a user needs (and uses) a resource for more than d days:

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

This is a boundary condition—a condition that makes requirements logically inconsistent. Boundary conditions can be expressed where two instantiations of a requirement compete or where an environmental condition conflicts with a requirement (i.e., an obstacle [Potts 1995]).

In KAOS, there are several ways to identify inconsistencies, as summarized in Table XII. An analyst manually applies the methods; however, some of the methods have been automated (e.g., Robinson [1994]).

Assertion regression (also known as, *goal regression* [Waldinger 1977]) is a particularly useful detection method. It is used to identify a boundary condition that leads to the divergence. In general, given rules of the form $X \Rightarrow Y$, regression determines what must be true if the rule is to be applied to satisfy a specific assertion A. Regression is essentially a backward application of a rule. It requires that the right-hand side of the rule, Y, be unifiable with the assertion, A; that is, the rule can assert A. To use assertion regression for divergence discovery, a requirement is negated. Next, regression generates the boundary condition. The preceding user and manager requirements illustrate this method:

(1) Negate one of the requirements. For example, the negated staff's requirement is $\Diamond\ \exists$ u:User, r:Resource Using(u,r) $\wedge\ \Box_{<\mathtt{d}}$ o Using(u,r).

(2) Now, the user's requirement can be rewritten for this specific context. The user's requirement becomes o Needs(u,r) $\rightarrow$ o Using(u,r) by universal instantiation and modus ponens.

(3) Finally, regressing the negated staff goal through the rewritten user goal yields the boundary condition, $\Diamond\ \exists$ u:User, r:Resource Using(u,r) $\wedge\ \Box_{<\mathtt{d}}$ o Needs(u,r).

In the preceding example, regression amounts to replacing the part of the goal that unifies with the right-hand side of the requirement with the left-hand side of the requirement. In general, unifying Y with part of an assertion $\neg A_i$ produces a match

Table XI. KAOS Inconsistency Types

Inconsistency	Description	Illustration
Process-level deviation	A state transition in the RE process that results in an inconsistency between an RE process rule and a state of the RE process.	Assigning responsibility for a goal (e.g., `InitiatorKnowsConstraints`) to two agents would violate a process-level rule that required single agent responsibility assignment. (See section 2.3.)
Instance-level deviation	A state transition in the running system that results in an inconsistency between a product level requirement and a state of the running system.	An individual, Jeff, failing to reply to a scheduler's request for updated constraints would violate the product-level requirement `InviteeResponds WithUpdatedConstraints`.
Terminology clash	A single real-world concept is given different syntactic names in the requirements.	Participant meeting attendance may be formalized two different ways: Attends(participant, meeting), Participates(participant, meeting)
Designation clash	A single syntactic name in the requirements specification designates different real-world concepts [Zave and Jackson 1997].	The designation, Attends(participant, meeting), may be interpreted differently: (a) "attending meeting m until the end," or (b) "attending part of meeting m."
Structure Clash	A single real-world concept is represented with different structures in the requirements specification.	The ExcludeDates attribute may be represented in two different ways: (a) SetOf[TimePoint], (b) SetOf[TimeInterval].
Conflict	A conflict among assertions occurs within a domain theory when (1) the set of assertions are logically inconsistent within the domain, (2) removing any one of the assertions removes the inconsistency.	Consider three views of a device's state: (1) InOperation $\Rightarrow$ Running (2) InOperation $\Rightarrow$ Running $\wedge$ Startup $\Rightarrow \neg$Running (3) $\Box$ InOperation
Divergence	A divergence among assertions occurs within a domain theory iff there exists a boundary condition B such that (1) the set of assertions become logically inconsistent within the domain when B is true, (2) removing any one of the assertions removes the inconsistency, and (3) there exists a feasible scenario S that satisfies B.	(1) `UserContinuesUseOfResource` (2) `UserDiscontinuesUseOfResource` B: $\Diamond$ ($\exists$ u′:User, r′:Resource) [Using(u′,r′) $\wedge$ $\Box_{\leq d}$ Needs(u′,r′)] See Section 5.4.2.
Competition	Competition is a particular type of divergence that occurs when different instances A[x_i] of the same universally quantified requirement x: A[x] are divergent.	Consider the scheduling goal: $\forall$m:Meeting, i:Initiator, p: Participant Requesting(i,m) $\wedge$ Invited(p,m) $\Rightarrow \Diamond$ ($\exists$d: Calendar) [m.Date = d $\wedge$ Convenient (d,m,p)] Two meetings, m and m′, can compete for a single convenient date.
Obstruction	An obstruction is a borderline case of divergence in that it only involves one assertion. The boundary condition amounts to an obstacle to the satisfaction of a requirement [Potts 1995].	The goal, `InformedParticipantsAttend`: $\forall$m:Meeting, p: Participant Invited(p,m) $\wedge$ Informed(p,m) $\wedge$ Convenient(d,m,p)] $\Rightarrow \Diamond$ Participates (p,m) is obstructed by the `LastMinute Impediment` obstacle: $\forall$m:Meeting, p: Participant Invited(p,m) $\wedge$ Informed(p,m) $\wedge$ Convenient(d,m,p)] $\wedge \Box$(IsTakingPlace(m) $\rightarrow \neg$Convenient (m.Date,m,p))

Table XII. KAOS Techniques for Inconsistency Detection

Method	Description	Illustration
Assertion regression	Given a divergence that involves assertions $\wedge_{1 \le i \le n} A_i$ and a set of domain properties, one can derive a boundary condition B by regressing the negation of an assertion $\neg A_i$ through the other assertions and domain properties.	Consider two goals of Section 5.4.2: `User ContinuesUseOfResource` and `User DiscontinuesUseOfResource`. Negate the second goal and regress it through the first. The resulting boundary condition has users needing a resource beyond its due date, thereby causing its use beyond the due date.
Pattern detection	Apply divergence patterns that have been proven to generate boundary conditions given certain patterns of assertions.	Given assertions of the *Achieve-Avoid* pattern: $(P \Rightarrow \Diamond Q) \wedge (R \Rightarrow \Box\neg S) \wedge (Q \Rightarrow S)$ Consider the boundary condition: $\Diamond (P \wedge R)$
Detection heuristics	Apply informal divergence heuristics that can suggest boundary conditions given certain types of goals.	If there is a `InformationGoal` and a `ConfidentialityGoal` concerning the same object, then consider a divergence between the two goals.

μ that can be substituted into $\neg A_i$ to produce the boundary condition, B. Thus, B applied to $X \Rightarrow Y$ yields $\neg A_i$.

Given a set of requirements, assertion regression can be applied to find various boundary conditions. Different boundary conditions can be had by (1) selecting different assertions in $\neg A_i$, (2) selecting different rules $X \Rightarrow Y$, (3) backchaining through the rules (e.g., $X \Rightarrow Y, W \Rightarrow X$), and (4) selecting different unifications in the case where there is more than one maximally general unification. Of course, it is critical to know which requirements to consider, as not all subsets of will have a divergence.

Pattern-based detection is another KAOS identification. Consider the following two goals [van Lamsweerde et al. 1998]:

```
Requirement  RequestSatisfied
Mode         Achieve
FormalDef
      ∀ u:User, r:Resource
      Requesting(u,r) ⇒ ◇Using(u,r)
Requirement  UnReliableResourceUsed
Mode         Avoid
FormalDef
      ∀ u:User, r:Resource
      ¬ Reliable(r) ⇒ □ ¬Using(u, r)
```

The two goals match the Achieve-Avoid pattern of Table XII, as follows: P: Requesting(u,r), Q: Using(u,r), R: ¬ Reliable(r), and S: Using(u,r). Applying the Achieve-Avoid pattern correctly generates the following boundary condition:

$\Diamond \exists$ u : User, r : Resource
Requesting(u, r) $\wedge$ ¬Reliable(r)

Finally, KAOS has detection heuristics that suggest where to look for boundary conditions. For example, given an `InformationGoal` and a `ConfidentialityGoal` involving the same object, consider a boundary condition for the object.

—*Focus*. Inconsistencies are not explicitly represented in KAOS. Thus, focusing on a subset of inconsistencies is not considered.

—*Resolution*. KAOS defines requirements resolutions methods, as summarized in Table XIII. (These are the formalized and specialized counterparts of the resolution methods found in Table VI, of Section 4.4.) An analyst manually applies the methods. As an example, Table XIII describes the Avoiding Boundary Conditions technique. It specifies that a divergence can be prevented by ensuring that the boundary condition is never satisfied. Applying this technique to the

Table XIII. KAOS Techniques for Resolution Generation

Method	Description	Illustration
Avoiding boundary condition	Given that a boundary condition B leads to a divergence, always avoid B.	Given a flight control system for which a rain-soak runway leads to a divergence (as in Section 2.2), then always avoid rain-soak runways.
Restore goal	If a boundary condition B cannot be avoided, then when it occurs restore the assertions involved in the divergence sometime thereafter.	The boundary condition derived for `UserContinuesUseOfResource` can be avoided by temporarily forcing the return of resource; that is, $\Diamond_{\leq d} \neg$ Using(u,r).
Anticipate conflict	Where a persistent condition P can eventually lead to conflict, anticipate the time period d that leads to conflict and introduce a new goal to negate the condition P before the time period elapses.	Consider a hospital patient monitoring system. If a monitored value exceeds its threshold for a long time (d), a patient could expire. So, prevent the persistent problem: "monitor working" $\wedge$ "monitored value exceeded" $\Rightarrow$ $\Diamond_{\leq d} \neg$ "monitored value exceeded"
Weaken goal	Weaken the goal (requirement) so that a boundary condition is not met, and thereby a divergence is removed. This can be done by "adding a disjunct, removing a conjunct, or adding a conjunct in the antecedent of an implication" [van Lamsweerde et al. 1998, p. 924].	As shown in Section 5.4.2, the goal, Requesting(u,r) $\Rightarrow \Diamond$ Using(u,r), has boundary condition, Requesting(u,r) $\wedge \neg$ Reliable(r). Incorporate the boundary condition into the goal: Requesting(u,r) $\wedge$ Reliable(r) $\Rightarrow \Diamond$Using(u,r)
Instantiate resolution patterns	Apply resolution patterns that have been proven to remove boundary conditions given certain patterns of assertions. For example, given that A must hold before d time units, extend the time bound to c, where c is greater than d.	Consider this goal sacrificing example: given two inconsistent goals, (a) Have(cake), and (b) Eat(cake), deleting goal (a) resolves the inconsistency.
Select alternative goal refinement	Given a goal G refined into subgoals, some of which being involved in a divergence, consider alternative refinements of G in which the diverging subgoals no longer appear.	As shown in Section 2.3, the goals `InitiatorKnowsConstraints` and `InitiatorNeverKnowsConstraints` have a boundary condition that can be resolved by substituting a scheduler for the initiator. This is an alternative refinement of the `ConstraintsKnown` goal.
Apply resolution heuristics	Apply informal resolution heuristics that can suggest how to manage boundary conditions given certain types of goals.	If there is a *Competition* divergence among agent instances, then consider the introduction of a reservation policy. A library's Reservation Desk is a common example.
Refine object	Given a boundary condition B that leads to a divergence, specialize an object into disjoint subtypes and restrict (via a conditional) the applicability of the divergent assertions into disjoint conditions (cf. Robinson and Volkov [1997]). This case-by-case approach applies different requirements to different object subtypes.	Consider three divergent scheduling goals, "Maintain schedule privacy," "Maintain schedule accuracy," and "Minimize participant effort." If the third goal is most important, then some refinements of the participant agent, such as a secretary, can view and maintain the schedule, but other agents cannot.

preceding resource boundary condition yields a new goal that limits requests to reliable resources:

```
Requirement   ReliableResources
Mode          Achieve
FormalDef
      ∀ u:User, r:Resource
      Requesting(u,r) ⇒
Reliable(r)
```

The other techniques of Table XIII similarly modify the requirements. For

example, Goal Restoration allows a goal to fail; however, afterwards, the conditions of the goal must be restored.

—*Selection*. Resolutions are not explicitly represented in the KAOS framework; thus resolution selection is not directly supported.

5.5. Deficiency-Driven Requirements Analysis

Since 1985, the University of Oregon requirements project members have published articles describing requirements analysis [Fickas 1985]. The group has relied on a AI techniques to construct automated assistants that critique requirements as part of a *deficiency-driven design* activity. KATE was the genesis, denoting *Knowledge-Based Acquisition of Specifications* [Fickas 1985]. Since then, various prototypes have been constructed under the KATE umbrella: Critic [Fickas and Nagarajan 1988], OPIE [Fickas and Anderson 1989], Oz[Robinson 1993; Robinson 1994], SC [Downing and Fickas 1991], and Critter [Fickas and Helm 1992].

In deficiency-driven design, violations of system requirements or environmental constraints drive the state-based search that generates alternative designs through the applications of design operators [Fickas and Helm 1992]. In KATE, a design is a detailed representation in which the interaction among selected constraints can be analyzed through simulation. The failure of a design to satisfy a requirement or constraint is characterized as a *deficiency*. Deficiencies are remedied by the application of operators. An operator may (1) add new agents, (2) reassign the responsibility for activities to different agents, (3) alter the communication among agents, or (4) weaken constraints. Operator application leads to a new design state, which can be analyzed for deficiencies. Design stops when all requirements are satisfied and there are no deficiencies.

5.5.1. Products. KATE projects do not create new requirements languages. Rather, they adapt existing representations (e.g., Numerical Petri Nets [Fickas and Helm 1992], Qualitative Physics [Downing and Fickas 1991], STRIPS predicates [Fickas and Anderson 1989]) to meet the needs of an automated assistant.

5.5.2. Activities. As part of the KATE project, computerized support has been specified and implemented. Collectively, the specified KATE tools provide considerable support to RIM activities:

—*Partitioning.* For the most part, KATE prototypes do not address the partitioning of requirements prior to analysis. Critter does form a decomposition of requirements through agent assignment (cf. Feather [1987]). Such a decomposition does aid analysis, but it has not been used to reduce the scope of analysis.

—*Identification.* Requirements interactions can be identified by simulation. The abstract planner, OPIE, can find a scenario of agent and environmental actions that lead to the satisfaction of a single requirement. Conversely, OPIE can find a failure scenario for a requirement. The conjunction of requirements can also be analyzed. In Oz, the conjunction of requirements held by different stakeholders was analyzed (using OPIE). If requirements conflicted, then Oz regressed the requirements through the scenario to identify the predicates that caused the conflict. Another KATE prototype, SC, demonstrated how qualitative interactions could be discovered using qualitative physics envisonment.[1] Finally, Critic demonstrated a case-based approach to interaction identification.

—*Focus.* Generally, conflicts are not explicitly represented in KATE. DealMaker is the exception. As a descendent of Oz, it defines an approach to focus on important conflicts. Its

[1] Qualitative physics envisonment is a method of qualitative simulation. Given qualitative constraints and behaviors as input, the simulation derives possible behavioral sequences; this process is called *envisionment* [Kuipers 1986].

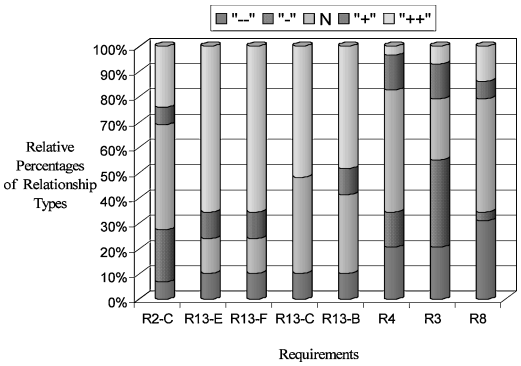

Fig. 8. An example of a root requirements interactions graph. The percentages show how each requirement interacts with all other requirements for five relationship types—from Very Conflicting (bottom) to Very Supporting (top).

Root Requirement Analysis supports (1) analysis of requirements interactions, and (2) ordering requirements by their degree of conflict [Robinson and Pawlowski 1998]. Using this information, one can iteratively resolve conflicts in a large requirements document. Root Requirements Analysis consists of four steps: (1) structure the requirements, (2) identify root requirements, (3) identify central interactions, and (4) iteratively resolve conflicts. A root requirement is an abstract requirement that implies significant interactions—typically, conflicts. After step 3, one can graph a requirement's contentiousness, which is the percentage of all conflicts in which the requirement participates. Figure 8 shows a contention graph. Notice that R8 and R3 are among the most contentious of all root requirements. Using such graphs, an analyst can guide analysis toward the central conflicts. By iteratively resolving the most contentious conflicts first, the number of conflicts can monotonically decrease, thereby providing an efficient resolution process [Robinson and Pawlowski 1998].

—*Resolution.* Oz, Critic, and Critter generate resolutions. Critter supports the Brinkmanship resolution heuristic that applies to a specific type of brink transition. A transition, T, is considered a brink transition if its application results in an inconsistent state. In Critter, a transition, T, is defined according to relations, P, over the vectors of a Numerical Petri Net [Wilbur-Ham 1985]. Thus, the following defines a brink transition:

$$\mathbf{T} \Rightarrow \mathtt{o}\ \neg\exists\,(\mathtt{x},\ \mathtt{y},\ \mathtt{z} : \mathtt{Vector})$$
$$\mathtt{P}(\mathtt{x},\ \mathtt{z}) \wedge \mathtt{P}(\mathtt{y},\ \mathtt{z}) \wedge \mathtt{x} \neq \mathtt{y}$$

A brink conflict occurs as the system transitions to a state in which the brink constraint fails. An instantiation of the brink conflict for a train control system follows:

$$\mathtt{StartTrain} \Rightarrow \mathtt{o}\,\exists(\mathtt{x},\ \mathtt{y} : \mathtt{Train},\ \mathtt{z} : \mathtt{Location})\,\mathtt{Location}(\mathtt{x},\ \mathtt{z}) \wedge \mathtt{Location}(\mathtt{y},\ \mathtt{z}) \wedge \mathtt{x} \neq \mathtt{y}$$

Given the (uncontrolled) StartTrain transition, it is possible to place two trains on the same block of track, possibly leading to a wreck. To resolve this conflict, the transition is replaced with a controlled transition: StartTrain-Controlled. In the controlled transition, if the brink condition can occur, then the controlled transition cannot execute; to ensure this, the analyst introduces a design fragment. Generalizing this resolution technique, we have the following:

Avoid Brink Transition
Given T, such that
$\mathbf{T} \Rightarrow \mathtt{o}\ \exists\ (\mathtt{x},\ \mathtt{y},\ \mathtt{z}\colon \mathtt{Vector})\ \mathtt{P}(\mathtt{x},\ \mathtt{z}) \wedge \mathtt{P}(\mathtt{y},\ \mathtt{z}) \wedge \mathtt{x} = \mathtt{y}$
Replace $\mathbf{T}$ with $\mathbf{T}' \equiv$
If $\mathtt{o}\exists\ (\mathtt{x},\ \mathtt{y},\ \mathtt{z}\colon \mathtt{Vector})\ \mathtt{P}(\mathtt{x},\ \mathtt{z}) \wedge \mathtt{P}(\mathtt{y},\ \mathtt{z}) \wedge \mathtt{x} \neq \mathtt{y}$ **Then T Else nil**

`Avoid Brink Transition` states that, if a transition can lead to a constraint failure in the next state, then incorporate the constraint as a condition of the transition.

Critic supports some conflict resolution. It recognizes certain design fragments as bad (i.e., should be absent), and others as good (i.e., should be present). It can map design fragments to requirements patterns. Thus, when it recognizes a design fragment it suggests elements to remove or add in order to satisfy the specified requirements, and remove the recognized conflicts.

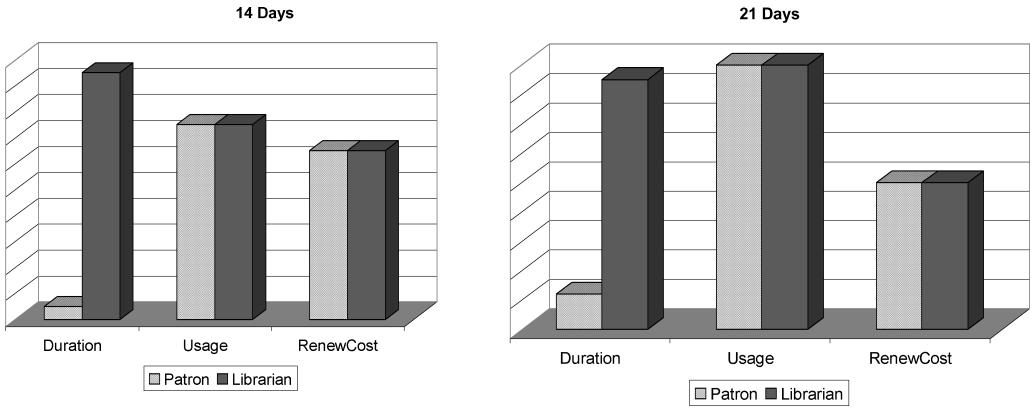

Fig. 9. Two bar charts illustrating the valuation of two different loan periods as perceived by a librarian and patron.

Finally, Oz supports many of the conflict resolution methods characterized in Section 5.4. A case study of DealMaker [Robinson and Volkov 1996, 1997], an Oz descendent, resulted in the following observations:

—*Automated resolution simplifies conflict resolution.* Once an analyst is familiar with the operations, application becomes routine.

—*Automated resolution can provide creative resolutions.* Contrary to expectations, the resolution operations routinely produced interesting resolutions that were not readily apparent.

—*Automated resolution can produce a large number of resolutions.* Resolution created, on average, 10 resolution types per conflict.

—*Resolutions of different conflicts often overlap.* Many of the conflicts involve the same objects, partly because of the common analysis used to derive the different views. Thus, resolution of one conflict can eliminate other conflicts.

—*Selection.* Oz and DealMaker support an interactive search procedure that (1) iteratively shows a multicriteria evaluation of resolution alternatives, and (2) allows for the application of new resolution methods. After an analyst is presented with conflicts, he or she can focus on a particular conflict, applying methods to generate resolutions.

Figure 9 illustrates the valuation of two resolutions. The resolutions concern a library loan policy conflict. The bar charts show the degree of satisfaction each participant receives for three criteria that have been associated with load period: renewal cost, usage, and duration. Thus, an analyst can see how each view values resolutions by observing the shading.

In an attempt to find a better solution, the analyst iteratively applies resolution methods. The overall result of this process is an AND/OR tree representing the resolutions explored. During the search process, the analyst need not explicitly define the satisfaction of each view. Instead, the analyst simply attends to alternatives that are deemed desirable, thereby implicitly settling on tradeoffs among views [Zeleny 1982].

5.6. Software Cost Reduction

Since 1978, the SCR project members from the Naval Research Laboratory have published articles describing software requirements and specification [Heninger et al. 1978]. Originally introduced to help specify the A-7E software package, the Software Cost Reduction (SCR) project has been successful in specifying and analyzing large, real-time, embedded systems. In SCR, a system and its environment are formally modeled. Then, using the SCR toolset, one can analyze

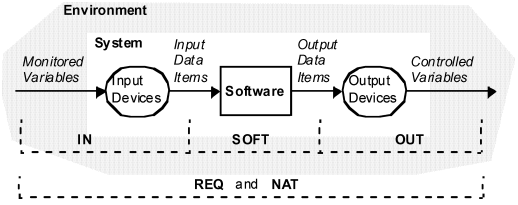

Fig. 10. The SCR four-variable model.

requirements for inconsistencies as well as check that specified properties hold (e.g., safety requirements).

SCR assumes a four-variable model, as illustrated in Figure 10 [Parnas and Madey 1995]. In this model, input devices watch *monitored variables* of the environment and produce values, called *input data items*, that are processed by the software. The software then produces *output data items* that are processed by output devices that affect *controlled variables* of the environment. The IN, SOFT, and OUT relations specify the mapping of monitored values to input values, input values to output values, and output values to controlled values, respectively. The REQ relation specifies the relationship of monitored variables to controlled variables. Finally, NAT describes natural laws, or constraints, of the environment.

An SCR requirements specification consists of (1) definitions for monitored, controlled, and intermediate *term* variables (2) tables of transformations that describe the software behavior (SOFT) by computing new variable values given changes to variables; and (3) properties of the system (REQ) and its environment (NAT). From such a description, the SCR toolset is able to check consistency within the requirements, as well as ascertain if specified properties (REQ) hold during the systems execution [Heitmeyer et al. 1996].

5.6.1. Products. The SCR language supports the four-variable model. Thus, it provides constructs to define variables, transformations, and properties. In SCR:

> a system Σ is represented as a 4-tuple, $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is the transform describing the allowed state transitions. In the initial version of the SCR formal model, the transform T is deterministic, i.e., a function that maps an input event and the current state to a (unique) new state. Each transition from one state to the next state is called a *state transition* or, alternately, a *step*. To compute the next state, the transform T composes smaller functions, called *table functions*. . . , These tables describe the values of the dependent variables—the controlled variables, the mode classes, and the terms. Our formal model requires the information in each table to satisfy certain properties. These properties guarantee that each table describes a total function. [Heitmeyer et al. 1996, p. 930]

Using SCR, the analyst defines transforms in a tabular form. Each table defines how an event will change a variable's value—shown as "mode" in the tables. An "*event* is a predicate defined on two consecutive system states that indicates a change in system state" [Heitmeyer et al. 1998b, p. 930]. It is denoted @T(c) $\equiv \neg$ c $\wedge$c′, where c is a condition that was false and then becomes true in the current state, denoted as c′.

As an example the consider a transition table shown in Table XIV for the variable `InjectionPressure` [Bharadwaj and Heitmeyer 1997]. This table defines when coolant can be injected into a pressurized container according to the container's water pressure.

The first row of Table XIV indicates that the mode of `InjectionPressure` changes from `TooLow` to Permitted when `WaterPressure` $\geq$ `Low` becomes true. A set of such tables concerning typed variables provides the basis for an SCR requirements specification.

Behavioral descriptions, such as those of Table XIV, can be enhanced with requirement properties, such as `@T(Water Pressure` $\geq$ `High)` $\Rightarrow$ `AlarmSounding`. It indicates that `AlarmSounding` shall be true when `WaterPressure` $\geq$ `High` is `True`. Such a requirement property does not specify the behavior of the software; rather it specifies a property that should always be true in the software (REQ). The SCR toolset can check the validity of such properties. SCR semantics describe properties

Table XIV. An SCR Transition Table for `InjectionPressure`

Old Mode	Event	New Mode
`TooLow`	`@T(WaterPressure` $\geq$ `Low)`	`Permitted`
`Permitted`	`@T(WaterPressure` $\geq$ `Permit)`	`High`
`Permitted`	`@T(WaterPressure` $<$ `Low)`	`TooLow`
`High`	`@T(WaterPressure` $<$ `Permit)`	`Permitted`

on states or between two states, but it does not have the full power of temporal logic, which also includes properties over sets of states, for example, $P \Rightarrow \Diamond Q$.

5.6.2. Activities. The SCR project has results in prototypes that provide considerable support to RIM activities:

—*Partitioning.* Given a requirement property, the SCR toolset can use a "program-slicing" technique to define only the relevant subset model of variables and transformations. Analysis on the reduced model can be much more efficient [Heitmeyer et al. 1996].

—*Identification.* The SCR toolset identifies two types of interactions [Heitmeyer et al. 1996]. First, inconsistencies among requirements can be found through static analysis of the requirements definition. The toolset can check that transformations are properly defined. This includes (1) disjoint definitions among table rows pairs for a transformation (i.e., $\wedge_{row \leq i \leq row+1}$ $Condition_i \Rightarrow False$); for example, if the third row of Table XIV were defined as @T(`WaterPressure` $\leq$ `Low`), then rows one and three would be nondisjoint; (2) complete coverage of a transformation by its table rows (i.e., $\vee_{1 \Box \leq i \leq \Box row}$ $Condition_i \Rightarrow True$); for example, if the first row of Table XIV were not included, then the transition from the mode TooLow would not be defined; (3) reachability of all of a variable's modes from its initial mode; for example, if the third row of Table XIV were not included, then the transition to the mode TooLow would not be defined. Second, the toolset can identify an interaction that exists between a requirement property and the software specification. It translates the SCR specification into a model checker, which explores all states in an attempt to show where the requirement property can fail [Holzmann 1997]. The resulting trace describes a scenario by which a requirement property can fail. During simulation of the trace, an analyst can observe failures in the highlighted SCR specification.

—*Focus, resolution, and selection.* Inconsistencies are not explicitly represented in SCR. Thus, focusing on a subset of inconsistencies or resolutions is not considered.

5.7. M-Telos

Two consecutive large European (ESPRIT) projects have focused on requirements engineering. From 1992 to 1995, the NATURE project (which stands for *Novel Approaches to Theories Underlying Requirements Engineering*) produced theories and tools for knowledge representation, domain engineering, and process engineering [Jarke et al. 1993, 1994]. The NATURE project led to the CREWS project, from 1996 to 1999. The CREWS (which stands for *Cooperative Requirements Engineering with Scenarios*) project developed, evaluated, and demonstrated the applicability of methods and tools for cooperative scenario-based requirements elicitation and validation [Jarke and Pohl 1994]. The results of both projects are too numerous to summarize here. Instead, we focus on one project, M-Telos, that defines a technique to manage requirements inconsistencies.

5.7.1. Products. As part of the CREWS project, M-Telos was defined as a means to manage multiple requirements views [Nissen and Jarke 1999]. The formalized framework provides an implementation that supports requirements development using a variety of notations, stakeholders views, or other requirements views.

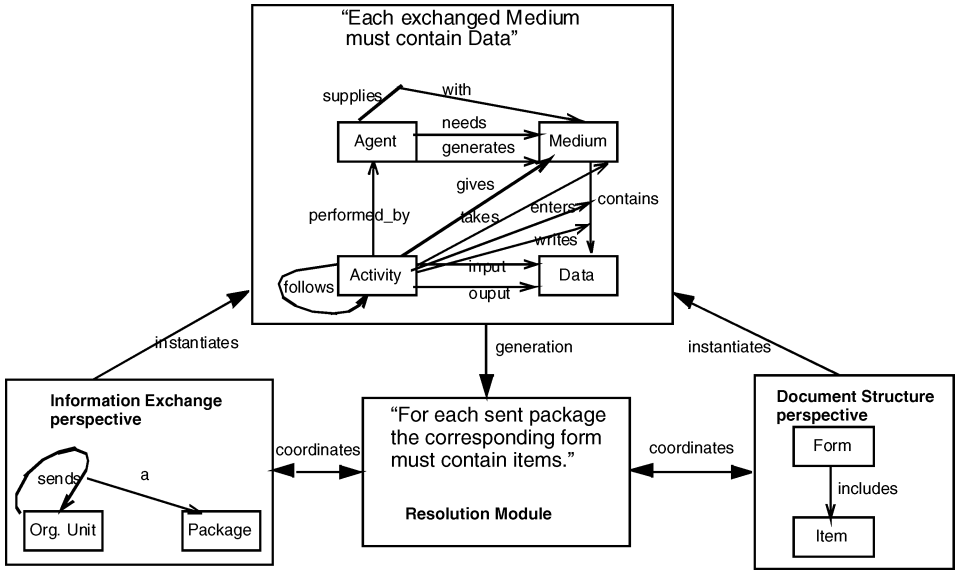

Fig. 11. An example illustrating perspective management in M-Telos.

As a means to support interview (and intraview) analysis, the framework supports the following techniques:

—separation of multiple partial models,
—dynamic definition and customization of notations,
—tolerance of conflicts,
—goal-orientated interview analysis, and
—dynamic definition and customization of analysis goals.

The M-Telos framework is implemented in ConceptBase, a deductive database that supports meta-modeling [Jarke et al. 1995].

Figure 11 illustrates the M-Telos approach to view management. In the approach, requirement notations are instantiated from a central meta-model (top of figure). Requirements views are defined in instantiated modules. Inconsistencies among modules are represented in *resolution* modules.

An *exchanged medium* goal was derived through stakeholder dialogs. In a business context, it means that documents exchanged should have information (i.e., they should not be content free). Figure 11 defines the goal as constraints on the meta-model; other specified constraints are not shown:

```
  Analysis Goal: ''Each exchanged
MEDIUM must contain DATA''
```

This is more formally represented in M-Telos as the following formula:

```
forall med//Medium, supp//Agent!
supplies, with//Agent!supplies!with
(with from supp) and (with to med)
==> exists data//Data, cont//Medium!
   contains
(cont from med) and (cont to data)
```

The `forall` quantification over `med//Medium` defines the `med` variable as an instance of `Medium`, which resides two instantiation levels above `med`.

The exchanged medium goal is an abstraction of the two requirements views (lower in Figure 11). For example, in the lower right module of Figure 11, requirements have the syntax: "`A FORM must include ITEMs.`" The goal abstracts the module, because a `FORM` is an instance of `MEDIUM` and `ITEM` is an instance of `DATA`. Similarly, the lower left module of Figure 11 instantiates the meta-model.

The resolution module of Figure 11 characterizes inconsistencies that can occur between requirements in the two modules.

M-Telos can automatically specialize the *exchanged medium* goal to address the combined context of the two modules:

Refined Goal: ``For each sent PACKAGE the corresponding FORM must contain ITEMs.''

Using the refined goal, the resolution module is able to monitor the other two modules. Every time a requirement instance is defined using the notation of either module, the resolution module checks for a violation of the refined goal.

Inconsistency checking in M-Telos can be conceptualized in terms of a stream of transactions that add or delete requirements. After a new requirement is added (or deleted), an analysis goal may become violated. This is called a *primary inconsistency* with respect to the goal. In the case of adding objects, those objects that were added and caused the inconsistency are considered *provisionally inserted objects*; deleting objects is analogous. As the result of a transaction, a goal's status may change from violated to satisfied. If the goal's satisfaction depends on objects that have been provisionally inserted (or deleted), then it is called a *secondary inconsistency* with respect to that goal. For example, assume a goal, g_1, is currently violated. Next, a requirement, r, is asserted; it satisfies g_1, but violates another goal, g_2. Thus, requirement r is provisionally inserted. Consequently, r creates a primary inconsistency with g_2, and a secondary inconsistency with g_1.

At the lowest level, inconsistency management becomes the task of computing goal satisfaction with respect to provisional objects. At a higher level, the user can specify the types of inconsistency that will be allowed: primary inconsistency (Allow/Not Allowed) and secondary inconsistency (Allow/Not Allowed). The combination provides for four levels of goal satisfaction. Grouping the two levels where some inconsistency is allowed gives us *goal satisfaction*, *qualified goal satisfaction*, and *goal violation*. Thus, the user of M-Telos can control what types of inconsistency will be tolerated, from no inconsistency to goal violations.

5.7.2. Activities. The M-Telos project provides support to RIM activities:

—*Partitioning.* The M-Telos framework allows for the partitioning of requirements into any subsets an analyst chooses. The framework itself does not provide any categories; rather, it provides the views into which one can place requirements. However, in related work, a subject's situation parameters (agent, focus, notation, and time) are proposed as means to systematically define bounded requirements sets [Motschnig-Pitrig et al. 1997].

—*Identification.* Once requirements and analysis goals are represented in M-Telos, the automated system can derive the inconsistencies incrementally or on demand. In related work, inconsistencies can be determined by comparing requirements against domain-independent analysis goals [Spanoudakis and Constantopoulos 1996; Spanoudakis and Finkelstein 1997], or domain-dependent analysis goals [Maiden and Sutcliffe 1994], rather than M-Telos's project-specific meta-model and goals.

—*Focus.* Inconsistencies are associated as goal violations. It has not been demonstrated how the framework could prioritize the further analysis or a resolution activity. However, goals for ordering goal analysis could be developed to specify the priority of consistency rules.

—*Resolution.* There is no resolution generation technique in M-Telos. Rather, analysis goals are used to identify inconsistencies, which are then represented in a resolution module. We believe that an interesting extension may be to show users provisional objects for qualified or violated analysis goals. This is similar to finding the causes of conflict through goal regression. With such an inconsistency context, users could generate alternative resolutions—for example, avoiding a boundary condition found among the provisional objects.

—*Selection.* Selection among resolutions is outside the scope of M-Telos.

6. RESEARCH OPPORTUNITIES

Although the seven projects just described support elements of RIM, much work remains to evolve RIM into a mature discipline. In this section, we present research opportunities for the five RIM activities and examine unifying approaches to making RIM more widely accepted.

6.1. Requirements Partitioning

A partitioning strategy suggests how to partition requirements to achieve a development goal, such as completeness, consistency, or efficiency. Although most of the seven RIM projects we have described have some partitioning support, it is not strategic. Typically, a database stores attributed requirements that support attribute-based retrieval. Ideally, a strategic tool would use the database to guide analysis toward requirement partitions containing the most significant requirements interactions. For example, in DealMaker (see Section 5.5) grouping requirements into classification hierarchies made conflict resolution more efficient for root requirements analysis [Robinson and Pawlowski 1998]. More strategies and tools are needed to assist the activities in all seven projects.

6.2. Interaction Identification

Most of the RIM projects do well at identifying requirements interactions. M-Telos does well at identifying logical requirement inconsistencies, for example, and WinWin effectively aids in identifying qualitative relationships according to an interaction model. Few projects track positive interactions, however. Traceability, in general, is a common part of requirements engineering [Gotel and Finkelstein 1995]. Tracing both the negative and positive interactions in which a requirement participates can facilitate interaction management. When resolving a conflict, for example, it may not be helpful to weaken a requirement that supports the satisfaction of many other requirements. Although the RIM projects effectively support the identification of individual interactions, they offer less support for identifying the complete set of requirements that either lead to an interaction or are affected by an interaction.

6.3. Interaction Focus

The RIM projects show mixed support for focusing on a subset of interactions. As with interaction identification, there is little support for addressing subsets of interactions, particularly in identifying dependencies among interactions. Which conflict, if resolved, will reduce the most dependent conflicts? Root requirements analysis addresses this problem [Robinson and Pawlowski 1998], and as interaction identification improves, interaction focus will become increasingly important.

6.4. Resolution Generation

Although *negotiation* is popular [Robinson and Volkov 1998], few of the RIM projects support the automated generation of creative resolutions. WinWin shows the analyst predefined text describing strategies to guide in conflict removal, and KAOS has formalized various methods, but only Oz [Robinson 1994] and DealMaker [Robinson and Volkov 1997], descendants of DDRA, provide tools that generate resolutions. A preliminary study indicated that automated resolution can be as successful as an analyst in generating many creative resolutions [Robinson and Volkov 1996]. Given the complexity of requirements documents, with the dependencies among requirements and requirements interactions, automated support for resolution generation would be well received.

6.5. Resolution Selection

Few RIM projects aid in resolution selection, leaving this decision process to the analyst. However, as tool support grows for the other RIM activities, analysts will become overwhelmed with analyzing the consequences of selecting particular

resolutions. Both Oz [Robinson 1994] and DealMaker [Robinson and Volkov 1997] build on work in decision support systems to partially automate resolution selection and DealMaker.

6.6. Incorporating Rim into Standard Practice

RIM is more than just individual techniques. It is the management of interactions that naturally occur as part of the development process. Research has clearly defined many individual techniques, but few projects integrate them, and no project has combined all aspects into a unified approach.

We see four areas that researchers must address to make RIM more widely accepted:

—*Strategies.* There are few guiding strategies for RIM. For example, one project has suggested that conflicts be tolerated [Nuseibeh 1996]. However, there is no description of how long they should be tolerated, what kind of conflicts should be tolerated, which conflicts should be resolved first, and how much conflict should be allowed in a document. The strategic application of RIM techniques, as a means to improve software development, has not been addressed adequately.

—*Integration.* Individual RIM techniques have been successful, but no project has addressed the total of RIM—from high-level informal strategies to low-level automated inconsistency identification. As a result, an analyst faced with RIM problems must attempt ad hoc to fill in the gaps of methodology and tool support.

—*Visualization.* RIM is about managing the complexity of interacting requirements. As tools are better able to detect and describe interactions, resolutions, and dependencies, analysts will increasingly bear the burden of understanding ever more complex analyses. A graphical user interface for visualizing and conducting RIM activities will be important.

—*Case studies and experiments.* Few RIM case studies or experiments have been conducted. At this early evolutionary stage of RIM, feedback from these evaluations will be critical in guiding the development of RIM techniques and application.

As researchers address these four areas, RIM will mature into a critical and integral area of requirements engineering. The results should be higher stakeholder satisfaction and fewer system failures.

ACKNOWLEDGMENTS

This article has been improved through the generous support of Steve Easterbrook, Martin Feather, Sol Greenspan, and Axel van Lamsweerde, who have commented on earlier drafts. Like most publications, this article has been improved in response to the critiques of the anonymous reviewers. For this article, the anonymous reviewers (specifically, Axel van Lamsweerde!) provided some of the most detailed and thoughtful remarks these authors have seen. Thank you! Nancy Talbert assisted in the editing. Thanks Nancy!

REFERENCES

AAAI. 1994. *Proceedings of the Workshop on Models of Conflict Management in Cooperative Problem Solving*. (Seattle, WA, Aug. (4). American Association for Artificial Intelligence, Menlo Park, CA.

ABRIAL, J. R. 1996. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge, U.K.

ADLER, M. R., DAVIS, A. B., WEIHMAYER, R., AND WORREST, R. W. 1989. *Conflict-Resolution Strategies for Nonhierarchical Distributed Agents*. Morgan Kaufmann, San Francisco, CA.

AINSWORTH, M., RIDDLE, S., AND WALLIS, P. J. L. 1996. Formal validation of viewpoint specifications. *Softw. Eng. J.* pp. 58–66.

ATLEE, J. M. 1993. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng.* 19, 24–40.

AVISON, D. E. AND WOOD-HARPER, A. T. 1990. *Multiview: An Exploration in Information Systems Development*. Blackwell Scientific Publications, Cambridge, MA.

BARBACCI, M., KLEIN, M. H., AND WEINSTOCK, C. B. 1995. Principles for evaluating quality attributes of software architecture. Tech. Rep. CMU/SEI-95-TR-021. Software Engineering Institute, Pittsburgh, PA.

BARBACCI, M., LONGSTAFF, T. H., KLEIN, M. H., AND WEINSTOCK, C. B. 1997. Quality Attributes.

Tech. Rep. CMU/SEI-96-TR-036. Software Engineering Institute, Pittsburgh, PA.

BARKI, H. AND HARTWICK, J. 1989. Rethinking the concept of user involvement. *MIS Quart. 13*, 1, 53–61.

BATINI, C., LENZERINI, M., AND NAVATHE, S. B. 1986. A comparative analysis of methodologies for database schema integration. *ACM. Comput. Surv. 18*, 4, 323–364.

BENDIFALLAH, S. AND SCACCHI, W. 1989. Work structures and shifts: An empirical analysis of software specification teamwork. In *11th International Conference on Software Engineering*. IEEE Press, Los Alamitos, CA, 260–270.

BENSALEM, S., LAKHNECH, Y., AND SAÏDI, H. 1996. Powerful techniques for the automatic generation of invariants. In *CAV'96—8th International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 1102 Springer-Verlag, Berlin, Germany, pp. 323–335.

BERNOT, G., GAUDEL, M. C., AND MARRE, B. 1991. Software testing based on formal specifications: A theory and a tool. *Softw. Eng. J. 9*, 6, 387–405.

BHARADWAJ, R. AND HEITMEYER, C. L. 1997. Model checking complete requirements specifications using abstraction. NRL Mem. Rep. NRL/MR/5540—97-7999. Naval Research Laboratory, Washington, DC.

BOEHM, B. 1981. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ.

BOEHM, B. AND EGYED, A. 1998. WinWin requirements negotiation processes: A multi-project analysis. In *Proceedings of the 5th International Conference on Software Processes*.

BOEHM, B., AND IN, H. 1996. Identifying quality-requirement conflicts. *IEEE Softw. 13*, 2 (March), 25–36.

BOEHM, B. W. 1988. A spiral model of software development and enhancement. *Comput. 21*, 61–72.

BOEHM, B. W. 1998. Seven basic principles of software engineering. *J. Syst. Softw. 3*, 1 (March), 3–24.

BOEHM, B. W., BROWN, J. R., KASPAR, H., LIPOW, M., MACLEOD, G. J., AND MERRITT, M. J. 1978. *Characteristics of Software Quality*. North-Holland, New York, NY.

BOEHM, B. W. AND ROSS, R. 1989. Theory W software project management: Principles and examples. *IEEE Trans. Softw. Eng. 15*, July, 902–916.

BOEHM, P. B., HOROWITZ, E., AND LEE, M. J. 1994. Software requirements as negotiated Win conditions. In *Proceedings of the First International Conference on Requirements Engineering*. IEEE. Press, Los Alamitos, CA.

BOTTEN, N., KUSIAK, A., AND RAZ, T. 1989. Knowledge bases: Integration, verification, and partitioning. *Eur. J. Operat. Res. 42*, 111–128.

BROOKS, F. P. 1987. No silver bullet: Essence and accidents of software engineering. Comput. *20*, 10–19.

BUCHANAN, B. G. AND SHORTLIFFE, E. H. 1984. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA.

CHECKLAND, P. 1981. *Systems Thinking, Systems Practice*. John Wiley & Sons, New York, NY.

CHEN H. L. K. J. 1992. Automatic construction of networks of concepts characterizing document databases. *IEEE Trans. Syst., Man Cybernet. 22*, 885–902.

CHEN H. L. K. J., BASU, K., AND NG, T. 1993. Generating, integrating, and activating thesauri for concept-based document retrieval. *IEEE Expert* (Special Series on Artificial Intelligence in Text-Based Information Systems) *8*, 25–34.

CHEN, M. AND NUNAMAKER, J. 1991. The architecture and design of a collaborative environment for systems definition. *Data Base*, *22*, 1/2 (Winter/Spring), 22–28.

CHIKOFSKY, E. J. AND RUBENSTEIN, B. L. 1993. CASE: Reliability engineering for information systems, in Computer Aided Software Engineering (CASE). *IEEE Expert* (Special Series on Artificial Intelligence in Text-Based Information Systems) *8*, 11–16.

CHRISTEL, M. G., WOOD, D. P., AND STEVENS, S. M. 1993. AMORE: The Advanced Multimedia Organizer for Requirements Elicitation. Tech. Rep. Software Engineering Institute, Pittsburgh, PA. CMU/SEI-93-TR-12.

CHUNG, L., NIXON, B., AND YU, E. 1994. Using quality requirements to systematically develop quality software. *Fourth Conference on Software Quality* (McLean, VA).

CHUNG, L., NIXON, B., AND YU, E. 1995. Using non-functional requirements to systematically support change. In *Proceedings of the Second International Symposium on Requirements Engineering*. IEEE Press, Los Alamitos, CA. 132–139.

CHUNG, L., NIXON, B., AND YU, E. 1996. Dealing with change: An approach using non-functional requirements. *Requirements Eng. J. 1*, 4, 238–260.

CHUNG, L., NIXON, B. A., YU, E., AND MYLOPOULOS, J. 1999. *Non-Functional Requirements in Software Engineering*. Kluwer, Boston, MA.

CLARKE, E. M., AND EMERSON, E. A. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 244–263.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press, Cambridge, MA.

CONRY, S. E., KUWABARA, K., LESSER, V. R., AND MEYER, R. A. 1991. Multistage negotiation for distributed satisfaction. *Trans. syst. man, Cybernet. 21*, 1462–1477.

CONRY, S. E., MEYER, R. A., AND LESSER, V. R. 1988. Multistage negotiation in distributed planning. In *Readings in Distributed Artificial Intelligence*, L. G. A. H. Bond, Ed. Morgan Kaufmann, San Meteo, CA, 367–384.

CORNFORD, S. L., FEATHER, M. S., KELLY, J. C., LARSON, T. W., SIGAL, B., AND KIPER, J. 2000. Design and development assessment. In *Proceedings of the 10th International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, CA, 105–114.

CROW, J., OWRE, S., RUSHBY, J., SHANKAR, N., AND SRIVAS, M. 1995. A tutorial introduction to PVS. In Proceedings of *WIFT'95*—Workshop on Industrial-Strength Formal Specification Techniques (Boca Raton, FL).

CUGOLA, G., DI NITTO, E., GUGGETTA, A., AND GHEZZI, C. 1996. A framework for formalizing inconsistencies and deviations in human-centered systems. *ACM Trans. Softw. Eng. Meth. 5*, 191–230.

DARDENNE, A., VAN LAMSWEERDE, A., AND FICKAS, S. 1993. Goal-directed requirements acquisition. *Sci. Comput. Programm. 20*, 3–50.

DARIMONT, R., DELOR, E., MASSONET, P., AND VAN LAMSWEERDE, A. 1998. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto) vol. 2. 58–62.

DARIMONT, R. AND VAN LAMSWEERDE, A. 1996. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the foundations of Software Engineering* (FSE'4, San Francisco, CA). ACM Press, New York, NY, 179–190.

DAVIS, A. 1993. *Software Requirements: Objects, Functions, and States*. Prentice Hall, Englewood Cliffs, NJ.

DELUGACH, H. S. 1992. Specifying multiple-viewed software requirements with conceptual graphs. *J. Syst. Softw. 19*, 207–224.

DELUGACH, H. S. 1996. An approach to conceptual feedback in multiple viewed software requirements modeling. In *Proceedings of Viewpoints 96: International Workshop on Multiple Perspectives in Software Development* (San Francisco, CA).

DOUGLAS, J. AND KEMMERER, R. A. 1994. Aslantest: A symbolic execution tool for testing ASLAN formal specifications. In *Proceedings of ISTSTA '94—International Symposium on Software Testing and Analysis*. ACM. Press, New York, NY, 15–27.

DOWNING, K., AND FICKAS, S. 1991. *A Qualitative Modeling Tool for Specification Criticism, Conceptual Modelling, Databases, and CASE: An Integrated View of Information Systems Development*. Ablex, Norwood, NJ.

DRUCKER, P. F. 1954. *The Practice of Management*. Harper & Row, New York, NY.

DUBOIS, E., BOIS, P. D., AND PETIT, M. 1993. Object-oriented requirements analysis: An agent perspective. In *ECOOP'93—7th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, Berlin, Germany, 458–481.

DURFEE, E. H. 1988. *Coordination of Distributed Problem Solvers*. Kluwer Academic Publishers, Boston, MA.

EASTERBROOK, S. 1993. Domain modeling with hierarchies of alternative viewpoints. In *Proceedings of the International Symposium on Requirement Engineering* (San Diego, CA) IEEE Press, Los Alamitos, CA, 65–72.

EASTERBROOK, S. 1994. Co-ordinating distributed ViewPoints: The anatomy of a consistency check. Concur. Eng.: Res. Appl. *2*, 209–222.

EASTERBROOK, S. AND NUSEIBEH, B. 1996. Using ViewPoints for inconsistency management. *Softw. Eng. J. 11*, 31–43.

EGYED, A. AND BOEHM, B. 1996. Analysis of software requirements negotiation behavior patterns. Tech. Rep., USC-CSE-96-504. University of Southern California, Los Angeles, CA.

EGYED, A. AND BOEHM, B. 1998. A comparison study in software requirements negotiation. *Proceedings of the 8th Annual International Symposium on Systems Engineering* (INCOSE'98).

EMMERICH, W., FINKELSTEIN, A., MONTANGERO, C., AND STEVENS, R. 1997. Standards compliant software development. In Proceedings of the *International Conference on Software Engineering Workshop on Living with Inconsistency*. IEEE Computer Societies Press, Los Alamitos, CA.

FEATHER, M. S. 1987. Language support for the specification and development of composite systems. *Trans. Program. Lang. Syst. 9*, 198–234.

FEATHER, M. S. 1989. Constructing specifications by combining parallel elaborations. *IEEE Trans. Softw. Eng. 15*, 198–208.

FEATHER, M. S., FICKAS, S., FINKELSTEIN, A., AND VAN LAMSWEERDE, A. 1997. Requirements and specification exemplars. *Automat. Softw. Eng. 4*, 4, 419–438.

FEATHER, M. S., FICKAS, S., VAN LAMSWEERDE, A., AND PONSARD, C. 1998. Reconciling system requirements and runtime behavior. In *Proceedings of the International Workshop on Software Specification and Design* (IWSSD'98). IEEE Computer Society Press, Los Alamitos, CA.

FESTINGER, L. 1964. *Conflict, Decision, and Dissonance*. Tavistock Publications, London, U.K.

FIADEIRO, J. L. AND MAIBAUM, T. 1995. Interconnecting formalisms: Supporting modularity, reuse and incrementality. In *Proceedings of the 3rd Symposium on the Foundations of Software Engineering*. ACM Press, New York, NY, 72–80.

FICKAS, S. 1985. A knowledge-based approach to specification acquisition and construction. Tech. Rep. CIS-TR-85-13. University of Oregon, Eugene, OR.

FICKAS, S. AND ANDERSON, J. 1989. A proposed perspective shift: Viewing specification design as a planning problem. In *Proceedings of the Fifth International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, CA, 177–184.

FICKAS, S. AND FEATHER, M. S. 1995. Requirements monitoring in dynamic environments. In *Proceedings of the 2nd International Symposium on Requirements Engineering* (York, England). IEEE Computer Society Press, Los Alamitos, CA, 140–147.

FICKAS, S. AND HELM, R. 1992. Knowledge representation and reasoning in the design of composite systems. *IEEE Trans. Softw. Eng. 18*, 6 (June), 470–482.

FICKAS, S. AND NAGARAJAN, P. 1988. Being suspicious: Critiquing problem specifications. In *Proceedings of the 7th National Conference on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 19–24.

FINKELSTEIN, A. AND FUKS, H. 1989. Multi-party specification. In *Proceedings of the 5th International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, CA, 185–195.

FINKELSTEIN, A., GABBAY, D., HUNTER, A., KRAMER, J., AND NUSEIBEH, B. 1994a. Inconsistency handling in multi-perspective specifications. *IEEE Trans. Softw. Eng. 20*, 569–578.

FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B., EDS. 1994b. *Software Process Modelling and Technology*. Research Studies Press Ltd. (Wiley), A theory of action for multiagent planning. Somerset, U.K.

FINKELSTEIN, A., KRAMER, J., NUSEIBEH, B., FINKELSTEIN, L., AND GOEDICKE, M. 1992. Viewpoints: A framework for multiple perspectives in system development. *Int. J. Softw. Eng. Knowl. Eng.* (Special Issue on Trends and Future Research Directions in SEE) *2*, 31–57.

FINKELSTIEN, A. E. 1996. Viewpoints 96: An international workshop on multiple perspectives in software development. ACM. In *Proceedings of the Symposium on the Foundations of Software Engineering* (San Francisco, CA), A. Finkelstien, Ed. ACM Press, New York, NY.

FISHER, R. AND WILLIAM, U. 1991. *Getting to Yes: Negotiating Agreement Without Giving In*. Penguin Books, New York, NY.

FOX, M. S., BARBUCEANU, M., AND GRUNINGER, M. 1996. An organisation ontology for enterprise modelling: Preliminary concepts for linking structure and behaviour. *Comput. Industry* 29, 123–134.

FRANCALANCI, C. AND FUGGETTA, A. 1997. Integrating conflicting requirements in process modeling: A survey and research directions. Inform. Softw. Tech. 39, 205–216.

GASSER, L. AND HUHNS, M. N. 1989. *Distributed Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

GEORGEFF, M. P. 1984. A theory of action for multiagent planning. In *Proceedings of 1984 Conference of the AAAI*. Morgan Kaufmann, San Mateo, CA, 121–125.

GILB, T. 1977. *Software Metrics*. Winthrop Publishers, Cambridge, MA.

GILB, T. 1988. *Principles of Software Engineering Management*. Addison-Wesley, Reading, MA.

GIRGENSOHN, A., REDMILES, D., AND SHIPMAN, F. 1994. Agent-based support for communication between developers and users in software design. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*. (KBSE'94, Monterey, CA). IEEE Computer Society Press, Los Alamitos, CA, 22–29.

GORDON, M. AND MELHAM, T. F. 1993. *Introduction to HOL*. Cambridge University Press, Cambridge, U.K.

GOTEL, O. AND FINKELSTEIN, A. 1995. Contribution structures. In *Proceedings of the 2nd International Symposium on Requirements Engineering* (RE'95). IEEE Computer Society Press, Los Alamitos, CA, 100–107.

GRAF, D. K. AND MISIC, M. M. 1994. The changing roles of the systems analyst. *Inform. Resources Man. J. 7*, 15–23.

GREENSPAN, S., MYLOPOULOS, J., AND BORGIDA, A. 1994. On formal requirements modeling languages: RML revisited. In *Proceedings of the Sixteenth International Conference on Software Engineering* (Sorrento, Italy). 135–148.

GRUNINGER, M. AND FOX, M. S. 1995. Methodology for the design and evaluation of ontologies. In *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing* (IJCAI'95, Montreal, P.Q., Canada).

GUSTAS, R. 1995. On related pragmatic categories and dependencies within enterprise modelling. In *Proceedings of the Second Scandinavian Research Seminar on Information and Decision Networks* (Vaxjo University, Sweden).

HAHN, U., JARKE, M., AND ROSE, T. 1991. Teamwork support in a knowledge-based information systems environment. *IEEE Trans. Softw. Eng. 17*, 467–482.

HALL, R. J. 1995. Systematic incremental validation of reactive systems via sound scenario generalization. *Automat. Softw. Eng. 2*, 131–166.

HALL, R. J. 1998. Explanation-based scenario generation for reactive system models. In *Proceedings of the Automated Software Engineering* (ASE'98). IEEE Computer Society Press, Los Alamitos, CA.

HAMMER, W. 1980. *Product Safety Management and Engineering*. Prentice-Hall, Englewood Cliffs, NJ.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16*, 403–414.

HAUSER, J. R. C. D. 1988. The house of quality. *Harvard Bus. Rev. 66*, 3, 63–73.

HEARST, M. 1998. Information integration. *IEEE Intell. Syst. 13*, 12–24.

HEIMDAHL, M. P. AND LEVESON, N. G. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Softw. Eng. 22*, 363–377.

HEIMDAHL, M. P. E. AND WHALEN, M. W. 1997. Reduction and slicing of hierarchical state machines. *ACM SIGSOFT Softw. Eng. Notes 22*, 450–467.

HEITMEYER, C., KIRBY, J., AND LABAW, B. 1998a. Applying the SCR requirements method to a weapons control panel: An experience report. In Proceedings of the Formal Methods in Software Practice'98 (Clearwater Beach, FL).

HEITMEYER, C. AND MANDRIOLI, D. 1996. Formal methods for real-time computing: An overview. In *Formal Methods for Real-time Computing*, C. Heitmeyer and D. Mandrioli, Eds. J. Wiley, Chichester, U.K., 1–32.

HEITMEYER, C. L., JEFFORDS, R. D., AND LABAW, B. G. 1996. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Method. 5*, 231–261.

HEITMEYER, C. L., KIRBY, J., JR., LABAW, B., ARCHER, M., AND BHARADWAJ, R. 1998b. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng. 24*, 927–948.

HEKMATPOUR, S. AND INCE, D. 1988. *Software Prototyping, Formal Methods, and VDM*. Addison-Wesley, Reading, MA.

HENINGER, K., PARNAS, D. L., SHORE, J. E., AND KALLANDER, J. W. 1978. Software requirements for the A-7E aircraft. Res. Rep., Naval Research Laboratory, Washington, DC.

HENINGER, K. L. 1980. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng. 6*, 2–13.

HEYM, M. AND H. OSTERLE. 1993. Computer-aided methodology engineering. *Inform. Softw. Tech. 35*, 345–353.

HOLZMAN, G. 1991. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ.

HOLZMANN, G. J. 1997. The model checker SPIN. *IEEE Trans. Softw. Eng. 23*, 279–295.

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating non-interfering versions of programs. *ACM Trans. Program. Lang. Syst. 11*, 345–387.

HUNTER, A. AND NUSEIBEH, B. 1998. Managing inconsistent specifications: Reasoning, analysis and action. *ACM Trans. Softw. Eng. Method. 7*, 4, 335–367.

JACKSON, D. AND DAMON, C. A. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng. 22*, 484–495.

JACKSON, M. J. 1995. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. ACM Press and Addison-Wesley, New York, NY, Wokingham, England, Reading, MA.

JACOBS, S. AND KETHERS, S. 1994. Improving communication and decision making within quality function deployment. In *Proceedings of the First International Conference on Concurrent Engineering, Research, and Application* (Pittsburgh, PA).

JANIS, I. L. AND MANN, L. 1979. *Decision Making : A Psychological Analysis of Conflict, Choice, and Commitment*. The Free Press, New York, NY.

JARKE, M., BUBENKO, J., ROLLAND, C., SUTCLIFFE, A., AND VASSILIOU, Y. 1993. Theories underlying requirements engineering: An overview of NATURE at Genesis. In *Proceedings of the First International Symposium on Requirements Engineering*. (San Diego, CA), 19–31.

JARKE, M., GALLERSDORFER, R., JEUSFELD, M. A., STAUDT, M., AND EHERER, S. 1995. ConceptBase—a deductive object manager for meta data management. *J. Intell. Inform. Syst. 4*, 167–192.

JARKE, M. AND POHL, K. 1994. *Requirements Engineering in 2001: (Virtually) Managing a Changing Reality*. Software Engineering Institute, Pittsburgh, PA, 257–266.

JARKE, M., POHL, K., DŠMGES, R., JACOBS, S., AND NISSEN, H. W. 1994. Requirements information management: The NATURE approach. *Ingenerie des Systemes d'Informations* (Special Issue on Requirements Engineering) *2*, 6, 609–636.

JEFFORDS, R. AND HEITMEYER, C. 1998. Automatic generation of state invariants from requirements specifications. In *Proceedings of the 6th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. (FSE'6), Lake Buena Vista, FL. ACM Press, New York, NY, 56–69.

JELASSI, M. T. AND FOROUGHI, A. 1989. Negotiation support systems: An overview of design issues and existing software. *Decision Support Syst. 5*, 2, 167–181.

JEUSFELD, M. A. AND JOHNEN, U. A. 1994. An executable meta model for re-engineering of database schemas. In *Proceedings of the 13th International Conference on Conceptual Modeling* (ER'94, Manchester, U.K.).

JOHANNESSON, P. AND JAMIL, M. H. 1994. Semantic interoperability—context, issues, and research directions. In *Proceedings of the Second International Conference on Cooperating Information Systems*. (Toronto, Ont. Canada).

JONES, C. 1995. Software challenges. *Comput. industry 28*, 10, 102–103.

JONES, C. 1996. *Patterns of Software Systems Failure and Success*. International Thomson Computer Press, London, U.K.

KANNAPAN, S. M. AND MARSKEK, K. M. 1993. An approach to parametric machine design and

negotiation in concurrent engineering. In *Intelligent Design and Manufacturing*, A. N. Kusiak, Ed. John Wiley and Sons, New York, NY, 509–534.

KANT, E. AND BARSTOW, D. 1981. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *Trans. Softw. Eng. SE-7*, 5, 458–471.

KARLSSON, J. AND RYAN, K. 1997. A cost-value approach for prioritizing requirements. *IEEE Softw. 14*, 5, 67–74.

KATZ, S., RICHTER, C. A., AND THE, K. S. 1987. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering*. (ICSE'87, Monterey, CA). 377–385.

KAZMAN, R., KLEIN, M. M. B., LONGSTAFF, T., LIPSON, H., AND CARRIERE, J. 1998. The architecture tradeoff analysis method. In *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS, Monterrey, CA). 68–78.

KECK, D. O. AND KÜHN, P. J. 1998. The feature and service interaction problem in telecommunications systems: a survey. *IEEE Trans. Softw. Eng. 24*, 779–796.

KERSTEN, G. E. AND SZPAKOWICZ, S. 1994. Negotiation in distributed artificial intelligence: Drawing from human experience. In *Proceedings of the 27th Annual Hawaii International Conference on Systems Sciences*. IEEE Press, Los Alamitos, 258–270.

KIM, E. AND LEE, J. 1986. An exploratory contingency model of user participation and MIS use. *Inform. Manage. 11*, pp. 87–97.

KLEIN, M. 1991. Supporting conflict resolution in cooperative design systems. *Trans. Syst. Man Cybernet. 21*, 1379–1390.

KLEIN, M. 2000. Towards a systematic repository of knowledge about managing collaborative design conflicts. In *Proceedings of the International Conference on AI in Design* (Boston, MA).

KOTONYA, G. AND SOMMERVILLE, I. 1996. Requirements engineering with viewpoints. *BCS/IEE Software Eng. J. 11*, 5–18.

KOYMANS, R. 1992. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Lecture Notes in Computer Science, vol. 651. Springer-Verlag, Berlin, Germany.

KRAUS, S. AND WILKENFELD, J. 1990. The function of time in cooperative negotiations. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, Mento Park, CA, vol. 1, 179–184.

KUIPERS, B. 1986. Qualitative simulation. *Art. Intell. 29*, 289–338.

KUMAR, K. AND WELKE, R. J. 1992. Methodology engineering: A proposal for situation-specific methodology construction. In *Challenges and Strategies for Research in Systems Development*, W. W. C. and J. A. Senn, Eds. John Wiley & Sons, London, U.K., 257–269.

KUSIAK, A. 1993. *Concurrent Engineering: Automation, Tools, and Techniques*. John Wiley & Sons, London, U.K.

KWA, J. B. 1988. Tolerant planning and negotiation in generating coordinated movement plans in an automated factory. In *Proceedings of the First International Conference on Industrial and Engineering Applications of Artificial Intelligence*.

LADKIN, P. 1995a. In *The Risks Digest* (online ACM moderated digest) *15*.

LADKIN, P. 1995b. In *The Risks Digest*, Neumann, Ed. *ACM SIGSOFT Softw. Eng. Notes 15*.

LAMSWEERDE, A. V. 2000. Formal specification: A roadmap, FOSE 00. In *The Future of Software Engineering*, A. Finkelstein, Ed. ACM Press, New York, NY, 147–159.

LAMSWEERDE, A. V. AND SINTZOFF, M. 1979. Formal derivation of strongly correct concurrent programs. *Acta Inform. 12*, 1–31.

LANDER, S. AND LESSER, V. R. 1989. A framework for the integration of cooperative knowledge-based systems. In *Workshop on Integrated Architectures for Manufacturing* (Detroit, MI).

LANDER, S. E. AND LESSER, V. R. 1993. Understanding the role of negotiation in distributed search among heterogeneous agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (Chambéry, France). 438–444.

LEMPP, P. AND RUDOLF, L. 1993. What productivity increases to expect from a CASE environment: Results of a user survey. In *Computer Aided Software Engineering (CASE)*, E. J. Chikofsky, Ed. IEEE Computer Society Press, Los Alamitos, CA, 147–153.

LEVENTHAL, N. 1995. Using groupware to automate joint application development. *J. Syst. Manage. 45*, 16–22.

LEVESON, N. 1995. *Safeware, System Safety and Computers*. Addison-Wesley, Reading, MA.

LEVESON, N. G., HEIMDAHL, M. P., AND HILDTRETH, H. 1994. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng. 20*, 684–706.

LIM, L. AND BENBASAT, I. 1992–1993. A theoretical perspective of negotiation support systems. *J. Manage. Inform. Syst. 9*, 27–44.

LIOU, Y. I. AND CHEN, M. 1993–1994. Using group support systems and joint application development for requirements specification. *J. Manage. Inform. Syst. 10*, 25–41.

LIU, F. X. AND YEN, J. 1996. An analytic framework for specifying and analyzing imprecise requirements. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE'18, Berlin, Germany). 60–69.

LUBARS, M., POTTS, C., AND RICHTER, C. 1993. A review of the state of practice in requirements

modeling. In *First International Symposium on Requirements Engineering*. IEEE Press, Los Alamitos, CA.

LYYTINEN, K. AND HIRSCHHEIM, R. 1987. Information systems failures—a survey and classification of the emperical literature. In *Oxford Surv. Inform. Tech. 4*, 257–309.

MAGAL, S. AND SNEAD, K. 1993. The role of causal attributions in explaining the link between user participation and information system success. *Inform. Resources Manage. J. 6*, 19–29.

MAIDEN, N., MINOCHA, S., RYAN, M., HUTCHINGS, K., AND MANNING, K. 1997. A co-operative scenario-based approach to the acquisition and validation of systems requirements. In *Proceedings of Human Error and Systems Development* (Glasgow University, Scotland).

MAIDEN, N. A. M. 1998. CREWS-SAVRE: scenarios for acquiring and validating requirements. *J. Automat. Softw. Eng. 5*, 4, 419–446.

MAIDEN, N. A. M. AND SUTCLIFFE, A. G. 1994. Requirements critiquing using domain abstractions. In *Proceedings of the International Conference on Requirements Engineering* (Colorado Springs, CO).

MALONE, T. W. AND CROWSTON, K. G. 1994. The interdisciplinary study of coordination. ACM *Comput. Surv. 26*, 87–119.

MANDRIOLI, D., MORASCA, S., AND MORZENTI, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst. 13*, 365–398.

MANNA, Z. AND GROUP, T. S. 1996. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *CAV'96—8th International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science, vol. 1102. Springer-Verlag, Berlin, Germany, 415–418.

MARKUS, L. AND KEIL, M. 1994. If we build it, they will come: Designing information systems that people want to use. *Sloan Management Review 35*, 4, 11–25.

MASSONET, P. AND VAN LAMSWEERDE, A. 1997. Analogical reuse of requirements frameworks. In *Proceedings of the 3rd International Symposium on Requirements Engineering (RE)*.

MATWIN, S., SZPAKOWICZ, S., KOPERCZAK, Z., KERSTEN, G. E., AND MICHALOWSKI, W. 1989. Negoplan: An expert system shell for negotiation support. *IEEE Expert 4*, 50–62.

MAZER, M. S. 1989. A knowledge-theoretic account of negotiated commitment. Tech. Rep. CSRI-237. University of Toronto, Toronto, Ont., Canada.

MAZZA, C., FAIRCLOUGH, J., MELTON, B., DE PABLO, D., SCHEFFER, A., AND STEVENS, R. 1994. *Software Engineering Standards*. Prentice Hall, Englewood Cliffs, NJ.

MCKEEN, J. AND GUIMARAES, T. 1997. Successful strategies for user Participation in systems development. *J. MIS 14*, 133–150.

MCMILLAN, K. L. 1992. Symbolic model checking—an approach to the state explosion problem. Tech. Rep. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

MCMILLAN, K. L. 1993. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, Boston, MA.

MI, P. AND SCACCHI, W. 1992. Process integration for CASE environments,. *IEEE Softw. 9*, 45–53.

MILLER, J., PALANISWAMI, D., SHETH, A., KOCHUT, K., AND SINGH, H. 1997. WebWork: METEOR's web-based workflow management system. Tech. Rep. Department of Computer Science, University of Georgia, Athens, GA.

MOFFETT, J. D. A. A. J. V. 2000. Behavioural conflicts in a causal specification. *Automat. Softw. Eng. 7*, 215–238.

MORGAN, C. 1990. *Programming from Specifications*. Prentice Hall, Englewood Cliffs, NJ.

MOSTOW, J. AND VOIGT, K. 1987. Explicit integration of goals in heuristic algorithm design. In IJCAI'87.

MOTSCHNIG-PITRIG, R., NISSEN, H. W., AND JARKE, M. 1997. View-directed requirements engineering: A framework and metamodel. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering* (SEKE'97, Madrid, Spain).

MULLERY, G. 1979. CORE—a method for controlled requirements expression. *Proceedings of the Fourth International Conference on Software Engineering*. In IEEE Computer Society Press, Los Alamitos, CA, 126–135.

MUMFORD, E. AND WEIR, M. 1979. Computer systems in work design—the ETHICS method. Associated Business Press, London, U.K.

MYLOPOULOS, J., BORGIDA, A., AND YU, E. 1997. Representing software engineering knowledge. *Automat. Softw. Eng. 4*, 291–317.

MYLOPOULOS, J., CHUNG, L., AND NIXON, B. 1992. Representing and using non-functional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng. 18*, 483–497.

MYLOPOULOS, J., CHUNG, L., AND YU, E. 1999. From object-oriented to goal-oriented requirements analysis. *Commun. ACM, 42*, 31–37.

NECHES, R., SWARTOUT, W., AND MOORE, J. D. 1985. Enhanced maintenance and explanation of expert systems through explicit models of their development. *Trans. Softw. Eng. SE-11*, 1337–1351.

NEUMANN, P. G. 1995. *Computer Related Risks*. Addison-Wesley, Reading, MA.

NISKIER, C., MAIBAUM, T., AND SCHWABE, D. 1989. A pluralistic knowledge-based approach to software specification. In *Proceedings of the ESEC'89—2nd European Software Engineering Conference*. Lecture Notes in Computer Science, vol. 387. Springer-Verlag, Berlin, Germany, 411–423.

NISSEN, H., JEUSFELD, A., JARKE, M., ZEMANEK, G., AND HUBER, H. 1996. Technology to manage

multiple requirements perspectives. *IEEE Softw. 13*, 2, 37–48.

NISSEN, H. W. AND JARKE, M. 1999. Repository support for multi-perspective requirements engineering. Inform. Syst. (Special Issue on Meta Modeling and Method Engineering) *24*, 131–158.

NORMAN, R. J. AND NUNAMAKER, J. F., JR. 1989. CASE productivity perceptions of software engineering professionals. *Commun. ACM 32*, 1102–1108.

NUSEIBEH, B. 1996. To be and not to be: On managing inconsistency in software development. In *Proceedings of the 8th International Workshop on Software Specification and Design* (IWSSD'8, Schloss Velen, Germany). IEEE Computer Society Press, Los Alamitos, CA, 164–169.

NUSEIBEH, B. AND FINKELSTEIN, A. 1992. ViewPoints: A vehicle for method and tool integration. In *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering* (CASE'92), Montreal, P.Q., Canada). IEEE Computer Society Press, Los Alamitos, CA, 50–60.

NUSEIBEH, B., KRAMER, J., AND FINKELSTEIN, A. 1994. A framework for expressing the relationship between multiple views in requirements specification. *IEEE Trans. Softw. Eng. 20*, 10, 760–773.

OLIVER, J. R. 1996. On artificial agents for negotiation in electronic commerce. In *Proceedings of the 29th Annual Hawaii International Conference on Systems Sciences*. IEEE Computer Society Press, Los Alamitos, CA, 337–346.

OLSEN, G. R., CUTKOSKY, M., TENENBAUM, J. M., AND GRUBER, T. R. 1994. Collaborative engineering based on knowledge sharing agreements. In *Proceedings of the 1994 ASME Database Symposium* (Minneapolis, MN).

OSTERWEIL, L. AND SUTTON, S. 1996. Using software technology to define workflow processes. In *Proceedings of the NSF Workshop on Workflow & Process Automation* (Athens, GA).

OSTERWEIL, L. J. 1987. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*. (Monterey CA, March). 2–13.

OWRE, S., RUSHBY, J., AND SHANKAR, N. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Softw. Eng. 21*, 107–125.

PARK, D. Y., SKAKKEBAEK, J., AND DILL, D. L. 1998. Static analysis to identify invariants in RSML specifications. In *Proceedings of FTRTFT'98—Formal Techniques for Real Time or Fault Tolerance*.

PARNAS, D. L. AND MADEY, J. 1995. Functional documentation for computer systems engineering. *Sci. Comput. Program. 25*, 41–61.

PERROW, C. 1984. *Normal Accidents: Living with High-Risk Technology*. Basic Books, New York, NY.

PERRY, D. E. AND WOLF., A. L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Softw. Eng. Notes, 17*, 40–52.

POHL, K. 1997. Requirements engineering: An overview. In *Encyclopedia of Computer Science and Technology*. A. Kent, and J. Williams, Eds. Marcel Dekker, New York, NY, vol. 36, suppl. 21.

POTTS, C. 1994. *Requirements Completeness, Enterprise Goals, and Scenarios*. Georgia Institute of Technology, College of Computing, Atlanta, GA.

POTTS, C. 1995. Using schematic scenarios to understand user needs. In *Proceedings of DIS'95—ACM Symposium on Designing Interactive Systems: Processes, Practices and Techniques* (University of Michigan, Ann Arbor, MI).

POTTS, C. AND BRUNS, G. 1988. Recording the reasons for design decision. In *Proceedings of the Seventh International Workshop on Software Specification and Design* (Singapore) 418–427.

POTTS, C., TAKAHASHI, K., AND ANTON, A. 1994. Inquiry-based requirements analysis. *IEEE Softw. 11*, 2, 21–32.

PRUITT, D. G. 1981. *Negotiation Behavior*. Academic Press, New York, NY.

QUEILLE, J. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in CAESAR. In *The Fifth International Symposium on Programming*. Lecture Notes in Computer Science, Vol. 137. Springer-Verlag, Berlin, Germany.

RAIFFA, H. 1968. *Decision Analysis*. Addison-Wesley, Reading, MA.

RAIFFA, H. 1982. *The Art and Science of Negotiation*. Harvard University Press, Cambridge, MA.

RAM, S. AND RAMESH, V. 1995. A Blackboard-based cooperative system for schema integration. *IEEE Expert/Intell. Syst. Appl. 10*, 56–62.

RAMESH, B. AND DHAR, V. 1992. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Trans. Softw. Eng. 18*, 6, 498–510.

RANGASWAMY, A., ELIASHBERG, J., BURKE, R. R., AND WIND, J. 1989. Developing marketing expert systems: An application to international negotiations. *J. Market. 53*, 24–39.

REUBENSTEIN, H. B. AND WATERS, R. C. 1991. The requirements apprentice: automated assistance for requirements acquisition. *IEEE Trans. Softw. Eng. 17*, 226–240.

RICHARDSON, D. J., AHA, S. L., AND O'MALLEY, T. O. 1992. Specification-based test oracles for reactive systems. In *Proceedings of the International Conference on Software Engineering*. (Melbourne, Australia). ACM Press, New York, NY. 105–118.

ROBBINS, S. P. 1983. *Organizational Behavior: Concepts, Controversies, and Applications*. Prentice Hall, Englewood Cliffe, NJ.

ROBEY, D., FARROW, D. L., AND FRANZ, C. R. 1989. Group process and conflict in systems development. *Manage. Sci. 35*, 1172–1191.

ROBINSON, W. N. 1989. Integrating multiple specifications using domain goals. In *Proceedings of the 5th International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, CA, 219–226. (Also available as Tech. Rep. CIS-TR-89-03, University of Oregon, Eugene, OR.).

ROBINSON, W. N. 1990. Negotiation behavior during requirement specification. In *Proceedings of the 12th International Conference on Software Engineering* (Nice, France). IEEE Computer Society Press, Los Alamitos, CA, 268–276. (Also available as Tech. Rep. CIS-TR-89-13, University of Oregon, Eugene, OR.).

ROBINSON, W. N. 1993. Automated negotiated design integration: Formal representations and algorithms for collaborative design. Tech. Rep., University of Oregon, Eugene, OR.

ROBINSON, W. N. 1994. Interactive decision support for requirements negotiation. *Concur. Eng.: Res. Appl.* (Special Issue on Conflict Management in Concurrent Engineering) *2*, 237–252.

ROBINSON, W. N. 1997. Electronic brokering for assisted contracting of software applets. In *Proceedings of the 30th Annual Hawaii International Conference on Systems Sciences*. IEEE Computer Society Press, Los Alamitos, CA, 449–458.

ROBINSON, W. N. 2002. Monitoring software requirements using instrumented code. In *Proceedings of the 35th Annual Hawaii International Conference on Systems Sciences*. IEEE Computer Society Press, Los Alamitos, CA.

ROBINSON, W. N. AND PAWLOWSKI, S. 1997. Surfacing root requirements interactions from inquiry cycle requirements. Tech. Rep. Georgia State University, Atlanta, GA.

ROBINSON, W. N. AND PAWLOWSKI, S. 1998. Surfacing root requirements interactions from inquiry cycle requirements. In *Proceedings of the Third IEEE International Conference on Requirements Engineering* (ICRE'98, Colorado Springs, CO). IEEE Computer Society Press, Los Alamitos, CA. 82–89.

ROBINSON, W. N. AND PAWLOWSKI, S. D. 1999. Managing requirements inconsistency with development goal monitors. *IEEE Trans. Softw. Eng. 25*, 816–835.

ROBINSON, W. N. AND VOLKOV, S. 1996. Conflict-oriented requirements restructuring. Tech. Rep. Georgia State University, Atlanta, GA.

ROBINSON, W. N. AND VOLKOV, S. 1997. A metamodel for restructuring stakeholder requirements. In *Proceedings of ICSE19—19th International Conference on Software Engineering* (Boston, MA). IEEE Computer Society Press, Los Alamitos, CA, 140–149.

ROBINSON, W. N. AND VOLKOV, S. 1998. Supporting the negotiation life-cycle. *Commun. ACM, 41*, 5, 95–102.

ROONG-KO, D. AND FRANKL, P. G. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Method. 3*, 101–130.

SANDHOLM, T. AND LESSER, V. 1995. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *Proceedings of the First International Conference on Multiagent Systems* (ICMAS'95, San Francisco, CA). 328–33.

SATHI, A., MORTON, T. E., AND ROTH, S. F. 1986. Callisto: An intelligent project management system, *AI Mag. 7*, 5, 34–52.

SCHNEIDER, F., EASTERBROOK, S. M., CALLAHAN, J. R., AND HOLZMANN, G. J. 1998. Validating requirements for fault tolerant systems using model checking. In *Proceedings of the Third IEEE Conference on Requirements Engineering* (Colorado Springs, CO). IEEE Press, Los Alamitos, CA.

SCHULER, D. AND NAMIOKA, A. 1993. Participatory Design. Lawrence Erlbaum Assoc., Hillsdale, NJ.

SHAKUN, M. F. 1991. Airline buyout: Evolutionary systems design and problem restructuring in group decision and negotiation. *Manage. Sci. 37*, 1291–1303.

SHAW, M. AND GAINES, B. 1989. Comparing conceptual structures: Consensus, conflict, correspondence and contrast. *Knowl. Acquis. 1*, 341–363.

SHAW, M. L. G. AND GAINES, B. R. 1988. A methodology for recognizing consensus, correspondence, conflict and contrast in a knowledge acquisition system. In *Proceedings of the Workshop on Knowledge Acquisition for Knowledge-Based Systems*. (Banff, Alta., Canada).

SHETH, A., ED. 1996. In *Proceedings of the Workshop on Workflow & Process Automation* (Athens, GA.). National Science Foundation, Washington, DC.

SMITH, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Comput. C29*, 1104–1113.

SOMMERVILLE, I. AND SAWYER, P. 1997. Viewpoints: Principles, problems and a practical approach to requirements engineering. *Ann. Softw. Eng. 3*, 21, 101–130.

SOMMERVILLE, I., SAWYER, P., AND VILLER, S. 1997. Viewpoints for requirements elicitation: A practical approach. Tech. Rep. CSEG/16/97. Lancaster University, Computing Department, Bailrigg, Larcs. U.K.

SPACAPPIETRA, S. AND PARENT, C. 1994. View integration: A step forward in solving structural conflicts. *IEEE Trans. Knowl. Data Eng. 6*, 258–274.

SPANOUDAKIS, G. AND CONSTANTOPOULOS, P. 1996. Analogical reuse of requirements specifications: A computational model. *Appl. Art. Intell.: An Internat. J. 10*, 281–306.

SPANOUDAKIS, G. AND FINKELSTEIN, A. 1995. Integrating specifications: A similarity reasoning approach. *Automat. Softw. Eng. J. 2*, 311–342.

SPANOUDAKIS, G. AND FINKELSTEIN, A. 1997. Reconciling requirements: A method for managing interference, inconsistency and conflict. *Ann. Softw. Eng. 3*, 21, 433–457.

STOREY, V. C., GOLDSTEIN, R. C., CHIANG, R. H. L., DEY, D., AND SUNDARESAN, S. 1997. Database design with common sense business reasoning and learning. *ACM Trans. Database Syst. 22*, 471–512.

SUTCLIFFE, A. G., MAIDEN, N. A. M., MINOCHA, S., AND MANUEL, D. 1998. Supporting scenario-based requirements engineering. *IEEE Trans. Softw. Eng.* (Special Issue on Scenario Management) *24*, 12, 1072–1088.

SYCARA, K. 1988. Resolving goal conflicts via negotiation. In *Proceedings of the AAAI'88*. American Association for Artificial Intelligence, Menlo Park, CA, 245–250.

SYCARA, K. 1991. Problem restructuring in negotiation. Manage. Sci. *37*, 1248–1267.

SYCARA, K., DECKER, K., PANNU, A., WILLIAMSON, M., AND ZENG, D. 1996. Distributed intelligent agents. *IEEE Expert/Intell. Syst. Appl. 11*, 36–46.

THOMPSON, J. M., HEIMDAHL, M. E., AND MILLER, S. P. 1999. Specification-based prototyping for embedded systems. In ESEC/FSE'99 (Toulouse, France). Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Berlin, Germany, 163–179.

VAN LAMSWEERDE, A., DARDENNE, A., DELCOURT, B., AND DUBISY, F. 1991. The KAOS project: Knowledge acquisition in automated specification of software. In *Proceedings of the AAAI Spring Symposium Series* (Stanford University, Stanford, CA). American Association for Artificial Intelligence, Menlo Park, CA, 59–62.

VAN LAMSWEERDE, A., DARIMONT, R., AND LETIER, E. 1998. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Trans. Softw. Eng. 24*, 908–926.

VAN LAMSWEERDE, A., DARIMONT, R., AND MASSONET, P. 1993. The meeting scheduler system—preliminary definition. Internal Rep., University of Louvain, Louvain, Belgium.

VAN LAMSWEERDE, A., DARIMONT, R., AND MASSONET, P. 1995. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Proceedings of the Second International Symposium on Requirements Engineering*. IEEE Computer Science Press, Los Alamitos, CA, 194–203.

VAN LAMSWEERDE, A. AND LETIER, E. 2000. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng. 26*, 978–1005.

VAN LAMSWEERDE, A. AND WILLEMET, L. 1998. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. Softw. Eng.* (Special Issue on Scenario Management) *24*, 12, 1089–1114.

VELTHUIJSEN, H. 1993. Distributed artificial intelligence for runtime feature-interaction resolution. *Comput. 26*, 48–55.

VESSEY, I. AND SRAVANAPUDI, A. P. 1995. CASE tools as collaborative support technologies. *Commun. ACM, 38*, 83–95.

VON MARTIAL, F. 1992. *Coordinating Plans of Autonomous Agents*. Springer-Verlag, Berlin, Germany.

WALDINGER, W. 1977. Achieving several goals semultaneously. In *Machine Intelligence*, E. Elcock and D. Michie, Eds. Ellis Horwood, Chichester, U.K., vol. 8, pp. 94–136.

WEINBERG, B. M. AND SCHULMAN, E. L. 1974. Goals and performance in computer programming. *Hum. Fact. 16*, 70–77.

WERKMAN, K. J. 1990a. Knowledge-based model of negotiationusing shareable perspectives. In *Proceedings of the 10th International Workshop on Distributed Artificial Intelligence* (Bandera, TX). American Association for Artificial Intelligence, Menlo Park, CA.

WERKMAN, K. J. 1990b. Knowledge-based model of using shareable perspectives. In *Proceedings of the Tenth International Conference on Distributed Artificial Intelligence* (Bandera, TX). American Association for Artificial Intelligence, Menlo Park, CA. 1–23.

WEYUKER, E., GORADIA, T., AND SINGH, A. 1994. Automatically generating test data from a Boolean specification. *IEEE Trans. Softw. Eng. 20*, 353–363.

WILBUR-HAM, M. C. 1985. *Numerical Petri Nets—A Guide*. Telecom Australia, Victoria, Australia.

WING, J. 1990. A specifier's introduction to formal methods. *IEEE Softw. 23*, 8–26.

WINOGRAD, T. AND FLORES, F. 1987. *Understanding Computers and Cognition*. Addison-Wesley, Reading, MA.

YAKEMOVIC, K. AND CONKLIN, J. 1990. Experience with the gIBIS model in a corporate setting. In *Proceedings of CSCW '90* (Los Angeles, CA).

YANG, W., HORWITZ, S., AND REPS 1992. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Trans. Softw. Eng. Method. 1*, 310–354.

YEN, J., LEE, H. G., AND BUI, T. 1996. Intelligent clearinghouse: Electronic marketplace with computer-mediated negotiation supports. In *Proceedings of the 29th Annual Hawaii International Conference on Systems Sciences*. IEEE Computer Society Press, Los Alamitos, CA, 219–227.

YEN, J. AND TIAO, W. 1997. A systematic tradeoff analysis for conflicting imprecise requirements. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering* (RE'97). IEEE Computer Science Press, Los Alamitos, CA.

YU, E. S. K. AND MYLOPOULOS, J. 1993. An actor dependency model of organizational work—with application to business process reengineering.

In *Proceedings of Conference on Organizational Computing Systems* (COOCS'93, Milpitas, CA).

ZAREMSKI, A. M. AND WING, J. 1997. Specification matching of software components. *ACM Trans. Softw. Eng. Method. 6*, 333–369.

ZAVE, P. AND JACKSON, M. 1993. Conjunction as composition. *ACM Trans. Softw. Eng. Method. 2*, 4, 379–411.

ZAVE, P. AND JACKSON, M. 1996. Where do operations come from? A multiparadigm specification technique. *IEEE Trans. Softw. Eng. XXII*, 508–528.

ZAVE, P. AND JACKSON, M. 1997. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Method. 6*, 1–30.

ZELENY, M. 1982. *Multiple Criteria Decision Making*. McGraw-Hill, New York, NY.

Received August 1999; revised July 2000; September 2001; December 2002; accepted April 2003