Managing Requirements Inconsistency with Development Goal Monitors

William N. Robinson and Suzanne D. Pawlowski
Department of Computer Information Systems
Georgia State University
Atlanta, GA 30301-4015 USA
+1 404 651 3867
wrobinson@gsu.edu **http://cis.gsu.edu/~wrobinso**

Abstract—Managing the development of software requirements can be a complex and difficult task. The environment is often chaotic. As analysts and customers leave the project, they are replaced by others who drive development in new directions. As a result, inconsistencies arise. Newer requirements introduce inconsistencies with older requirements. The introduction of such requirements inconsistencies may violate stated goals of development. In this article, techniques are presented that manage requirements document inconsistency by managing inconsistencies that arise between requirement development goals and requirements development enactment.

A specialized development model, called a requirements dialog meta-model, is presented. This meta-model defines a conceptual framework for dialog goal definition, monitoring, and in the case of goal failure, dialog goal reestablishment. The requirements dialog meta-model is supported in an automated multi-user World Wide Web environment, called DEALSCRIBE. An exploratory case study of its use is reported.

This research supports the conclusions that: 1) an automated tool that supports the dialog meta-model can automate the monitoring and reestablishment of formal development goals, 2) development goal monitoring can be used to determine statements of a development dialog that fail to satisfy development goals, and 3) development goal monitoring can be used to manage inconsistencies in a developing requirements document. The application of DEALSCRIBE demonstrates that a dialog meta-model can enable a powerful environment for managing development and document inconsistencies.

Index Terms—**Requirements engineering, inconsistency and conflict management, process modeling and monitoring, meta-modeling, CASE.**

 © 1999 William N. Robinson

Managing Requirements Inconsistency with Development Goal Monitors

I. Introduction

Requirements engineering can be characterized as an iterative process of discovery and analysis, designed to produce an agreed-upon set of clear, complete, and consistent system requirements. The process is complex and difficult to manage, involving the surfacing of stakeholder views, developing of shared understanding, and building of consensus. A key challenge facing the analyst is the management and analysis of a dynamic set of requirements as it evolves throughout this process. Techniques have been developed to support aspects of this process; however, support of *requirements development monitoring* has been lacking.

Requirements development monitoring entails: 1) the specification of requirement development goals, and 2) the creation development goal monitoring agents. As a requirements development unfolds, monitoring agents provide warnings, or even remedies, when development goals are not satisfied. Such monitoring is similar to the performance monitoring repeatedly applied to software as part of run-time optimization. However, requirements development monitoring differs in that: 1) it applies over the (longer) life-time of a requirements development, and 2) monitored goals often refer to complex interrelationships among the development products over time.

Analysts can benefit from requirements development monitoring. Since requirements development often involves changing analysts, stakeholders, and requirements, managing the requirement development can be a challenge. Monitoring addresses this challenge by providing analysts with: 1) notifications when the development *does not* satisfy development goals, and 2) guarantees when the development *does* satisfy development goals.

This research defines an approach to requirements development monitoring. It offers an incremental improvement to the state of the art in process monitoring. It shows how formal development goals can be translated into software monitored goals. It is based on a formal dialog meta-model. Once dialog statement types and development goals are formally specified in terms of the meta-model, a dialog support system, including goal monitoring agents, can be automatically created. A software tool was constructed to demonstrate the effectiveness of the approach. Additionally, a case study was conducted to demonstrate the practicality of this meta-modeling approach to requirements development monitoring.

A. Managing Requirements Dialog

Stakeholder dialog is a pillar of the requirements development process. Techniques have been developed to facilitate dialog (e.g., JAD, prototyping, serial interviews) and to document and track requirements as they evolve (e.g., CASE). A requirements dialog can be viewed as a series of conversations among analysts, customers, and other stakeholders to develop a shared understanding and agreement on the system requirements. Typically, the analyst converses with the customers about their needs. In turn, the analyst may raise questions about the requirements, which lead to further conversations. Within the development team, analysts will also converse among themselves about questions that arose during their analysis of the requirements—sometimes the result of sophisticated analysis; other times, the result of simply reading two different paragraphs in the same requirements document.

Like many dialogs, requirements development can be difficult to manage. Empirical studies have

documented the difficulties and communication breakdowns that are frequently experienced by project teams during requirements determination as group members acquire, share, and integrate project-relevant knowledge[30][72]. Requirements or their analyses are forgotten. Different requirements concerning the same objects arise at different times. Inconsistency, ambiguity, and incompleteness are often the result.

The objective of this research is to address the monitoring of requirement dialog goals—especially requirements consistency development goals. Requirements inconsistency is a critical driver of the requirements dialog. If one can manage requirements inconsistency, one can manage a key driver of complexity and confusion in the requirements dialog. Goal monitoring simplifies inconsistency management by alerting analysts of important events. Analysts can filter the chaotic activities of a requirement dialog through goal monitors and focus their attention on important changes.

In this article, we present our meta-modeling approach to requirements development monitoring (§II). A description of a supporting software tool (§III), called DEALSCRIBE, illustrates how the meta-model can provide for automated support. The requirement development protocol of section IV illustrates how development goals can be expressed as instances of the dialog meta-model. That protocol, for Root Requirements Analysis, can be monitored by DEALSCRIBE. An exploratory case study of its application is presented in section V. The case study illustrates the practical utility of the meta-model. It shows that a tool, such as DEALSCRIBE, can support development goal monitoring as defined by the dialog meta-model. Finally, we conclude (§VI) that the dialog meta-model can provide analysts with automation, assurance, and understanding that facilitates the management of inconsistencies that arise during multi-stakeholder requirements dialog. The remainder of this introduction motivates this research and places it in context.

B. Inconsistency and Conflict: Drivers of Requirements Dialog

Requirements engineering is difficult. Like software engineering[6], these difficulties have been characterized as essential and accidental[19]. Essential difficulties are inherent characteristics of the requirements engineering task; they include: comprehension, communication, control, and dependencies. Accidental difficulties are characteristics that arise from the imperfect practice of requirements engineering; they include: post project requirements construction, confused requirements audience or purpose, and poor technical quality.

Dialog among system stakeholders addresses essential difficulties of requirements engineering. Through communications, stakeholders improve their comprehension of requirements and their dependencies. Through management operations, such as versioning, change is controlled. However, throughout this process there is a more fundamental dialog driver.

Inconsistency[51], conflict, break-down[74], cognitive dissonance[21]—these are terms that characterize aspects of uncovering unexpected ideas during problem solving. The general concept of "conflict" has been characterized as a key driver of group communication and productive work[58]. Conflict has been empirically shown to be a driver of systems development[36][40][59] and, more specifically, requirements development[4][28][37][65].

Two basic forces give rise to requirements "conflicts". 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"[16]. Second, the social nature of constructing a requirements document gives rise to *conflict*—requirements held by two or more stakeholders that cause an inconsistency.

Technical and social forces give rise to inconsistencies and conflicts that drive essential difficul-

ties of requirements engineering. By managing inconsistencies and conflicts, one can manage a key aspect of requirements engineering. Specific problem types will illustrate.

Consider three technical problems that lead to requirements inconsistency:

- **Voluminous requirements.** The shear size of a requirements document can lead to inconsistency, such as varied used of terminology. This is especially true as requirements are modified.
- **Changing requirements and analysts.** As a requirements document is developed, new requirements are added, older ones are updated. One change request can lead to a cascade of other change requests until the requirements reach a more consistent state. As a result, the document is typically in a transitory state where many semantic conflicts exist, of which most are expected to be resolved simply by bringing them to the current state as (implicitly) understood by the analysts. Unfortunately, the implicit current state of requirements is lost when analysts leave a long term project. Moreover, requirement concepts and their expressions will vary with the composition of team members. Such changes introduce inconsistencies.
- **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. Implicit requirement dependencies often hide requirements inconsistencies.

Consider three social problems that lead to requirement conflicts:

- **Conflicting stakeholder requirements.** Different stakeholders often seek different requirements that cannot be mutually achieved. This problem is exacerbated by changing stakeholders.
- **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; however, such access can be difficult to come by[35]. Moreover, one department of an organization may claim to be "the" customer; however, when it comes to the final purchase decision, it may be another department[35]. Such organizational interactions can lead to drastic changes in the 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 have a positive effect on user satisfaction of development products[3]. User participation is particularly effective during requirements development[28][33][34][37][42].

Requirements management attempts to address such technical and social problems as part of a strategy to manage the inconsistencies and conflicts that contribute to the essential difficulties of requirements engineering. Requirements management can address many of the problems by supporting requirements tractability in a dynamic, multi-stakeholder environment. For example, by tracking the statements asserted by analysts and stakeholders as they enact a requirements dialog, one 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, one can find trends (e.g., convergence or divergence) of expectations. A requirements management tool can even support a dialog protocol that aids in the detection and resolution of multi-stakeholder requirement conflicts, as we show in section IV.

C. A Need to Support Analysts in Inconsistency Management

Requirements analysts need tools to assist them in reasoning about requirements. To some degree, Computer Aided Software Engineering tools have been successful in providing support for

modeling and code generation[9][32][50]; however, they have been less successful in supporting requirements analysis[32]. In fact, the downstream life-cycle successes of these tools may be one of the reasons that systems analysts spend a greater percent of the time on requirements analysis than ever before[24]. Thus, analysts will benefit from techniques and tools that directly address requirements analysis.

A significant part of requirements analysis concerns the identification and resolution of requirements faults. Faults include: incorrect facts, omissions, inconsistencies, and ambiguities [48]. Many current research projects are aimed at identifying such faults from requirements. These include: model checkers, terminological consistency checkers, knowledge-based scenario checkers; additionally, more generic tools, such as simulation and visualization, are available to requirements analysts. For the most part, these tools are cousins of similar tools applied to programming languages that check for syntactic errors or perform checks of program inputs and path execution. However, requirement faults are rarely traced back to the original stakeholders, nor has there been much support for resolving such faults. Yet, there is still a belief that conflict identification and resolution are key in systems development[36][59].

Empirical studies of software development projects have identified a need for issue tracking tools[12][73]. Typical problems include: 1) unresolved issues that do not become obvious until integration testing, and 2) a tendency for conflicts to remain unresolved for a period of time. Inadequate tools for tracking issue status (e.g., conflicting, resolved) was identified as a great concern to practicing system engineers.

Collaborative CASE tools may provide an answer. Collaborative CASE aims to support task, team, and group level analysis by addressing information control, sharing, and monitoring[68]. However, many of these tools still fall short in their management of the requirements elicitation and development process[34]. Davey notes, "current implementations of CASE tool technology encourage an individual approach to work, even if they provide multi-user capability. However, analysts improve the quality of their work by discussing and reviewing it with colleagues. CASE tools are useful, but they are no substitute for interactive group discussion. Until more responsive interfaces are available we will only be able to make limited use of them."—p. 15[14].

D. Research Addressing Requirements Management

There is a growing literature on requirements inconsistency management. Fickas and Feather proposed requirements monitoring to track the achievement of requirements during system execution as part of an architecture to allow the dynamic reconfiguration of component software[22]. Feather has produced a working system, called FLEA, that allows one to monitor events defined in a requirements monitoring language[20]. Emmerich *et al.* have illustrated how the technique may be used to monitor process compliance[18]; for example, compliance to ISO 9000 or IEEE process descriptions[41]. Our work on dialog monitoring is derived from these works, but also includes an element of dialog structuring.

Two projects explicitly address requirements dialog structures. First, Chen and Nunamaker have proposed a collaborative CASE environment, tailoring GroupSystems decision room software, to facilitate requirements development[11]. Using C-CASE, one can track and develop requirements consensus. Second, Potts *et al.*, have defined the Inquiry Cycle Model of development to instill some order into requirements dialogs[53]. Requirements are developed in response to discussions consisting of questions, answers, and assumptions. By tracking these types of dialog elements, dialog is maintained, and inconsistency, ambiguity, and incompleteness are kept in check through specific development operations (e.g., scenario analysis).

Workflow and process modeling provide some solutions for the management of requirements development[66]. It is possible, for example, to generate a work environment from a hierarchical multi-agent process specification[44]. There has been some attempt to incorporate such process models into CASE tools[43]. However, these tools generally aid process enactment, through constraint enforcement. However, as Leon Osterweil notes:

> 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.[52]

In support of a reactionary approach, the dialog meta-model eschews process enforcement and supports the expression and monitoring of process goals.

Other projects indirectly address the management of requirements inconsistency. These include: 1) an ontological approach, in which conflict surfacing is assisted by providing a set of meaningful terms, or ontology, by which one can specify conflict relationships between requirements[10][55][75]; 2) a methodological approach, in which the application of a system development method surfaces conflicts—for example, CORE[46], ETHICS[47], Soft Systems Method[8], ViewPoints[51], KAOS[13], and CORR[60]; and 3) a technological approach in which a specific technique can be used to surface requirements conflicts—for example, conflict detection through a collaborative messaging environment[5][25][31], structure-based conflict detection[67], scenario-based conflict surfacing[2][38][53], plan-based reasoning[69], and conflict classification[15][29].

The dialog meta-model is neutral to the above approaches. A methodology, conflict ontology, or automated techniques can be defined as instances of it. The DEALSCRIBE implementation provides automated support for the enactment of a structured dialog which is defined as an instance of the dialog meta-model.

E. Support of Development Goal Monitoring

Goal monitoring addresses technical and social forces that give rise to requirements inconsistencies and conflicts by managing the changes that directly affect requirements. Changes in stakeholders, analysts, requirements, or analyses are tracked as part of the dialog meta-model. For example, a software development organization may seek equal contribution from its stakeholders. The dialog goal, NEARLYEQUALCONTRIBUTION, captures this.

NEARLYEQUALCONTRIBUTION ≡ Each stakeholder must contribute a nearly equal number of statements to the requirement dialog

The NEARLYEQUALCONTRIBUTION goal reflects a social development goal that is monitored through the relative number of statements contributed by stakeholders.

Requirement traceability enables the monitoring of the NEARLYEQUALCONTRIBUTION goal. Gotel and Finkelstein define requirements traceability as “the ability to describe and follow the life of a requirement, in both a forward and backward direction”[23]. To monitor the NEARLYEQUALCONTRIBUTION goal, there must be backward traceability from the requirement to the contributing stakeholder. A framework that tracks development objects, and the agents and operations that act on them, supports traceability[57]. Goal monitoring supports such traceability.

Goal monitoring brings traceability to life through its notifications. As development occurs, stakeholders receive feedback on interesting events; for example, warnings on the violation of the NEARLYEQUALCONTRIBUTION goal. Goal monitors can provide feedback on changing or inconsistent requirements. As this becomes apparent during development, analysts may become aware of more social problems, such as conflicting stakeholder requirements.

F. Problems of Development Goal Monitoring

To define a system that supports development goal monitoring, three main problems must be addressed.

1) *How can development goals be specified?* Development goals describe relationships among the stakeholders, products, and processes of the development dialog. A goal specification language should facilitate such expressions.
2) *How can development goals be monitored?* As the development dialog is enacted, events occurring in the dialog may result in goal failure. Goal monitoring should detect such goal violations as they occur.
3) *How can violated development goals be restored?* When a development goal is violated, a goal monitoring system should facilitate the automated reestablishment of the goal.

The following section presents an approach that answers each of these questions.

II. A Dialog Support System and Meta-Model

Our requirements dialog support system was designed to provide solutions to the problems of development goal monitoring, as well as address basic needs of requirement dialog support. The needs include the following.

- The need to represent multiple stakeholder requirements, even if conflicting.
- The need to identify and understand requirements interactions.
- The need to track and report on development issues.
- The need to develop shared understanding and consensus through requirements analysis and negotiation.
- The need to support dynamic, dialog-driven requirements development.

Stakeholder dialog is central to these needs. Additionally, analysts must be able to analyze the developing requirements. The needs can be supported through a dialog support system and its meta-model.

We have defined a *dialog support system* and its *dialog meta-model*. As illustrated in figure 1, the dialog system regards a dialog as a stream of statements. Each statement is either passive information or an active operation. Statements are instances of the dialog statement model. A dialog is

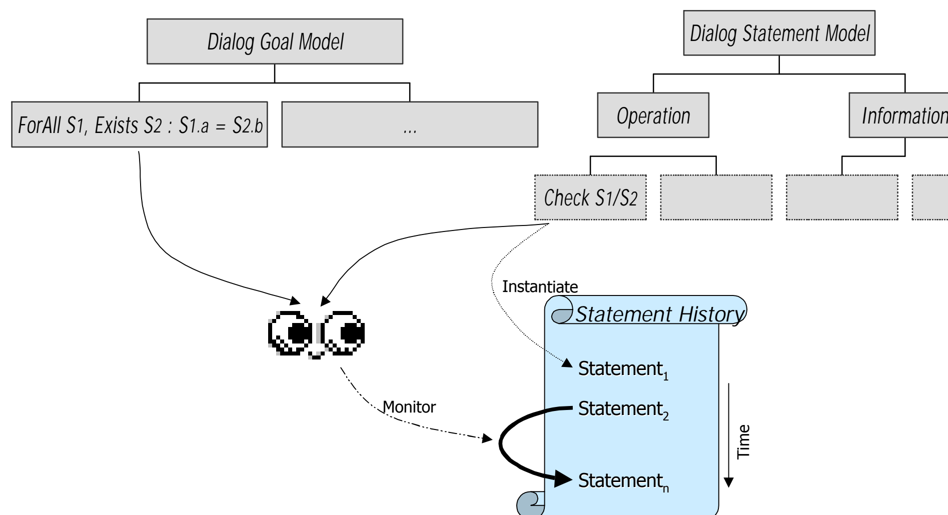


Fig 1. An illustration of an application of the dialog meta-model.

a continual stream of statements from multiple stakeholders; some statements initiate new ideas, others are in response to previous statements. As the dialog expands, dialog goals can be compared against the dialog to determine their status.

The meta-model can be instantiated to define a typical process model, with a distinction of process and product. Consider information statements as a products, operation statements as actions, and dialog goals as defining a process model. As such, the dialog meta-model is a process model with an explicit representation of the process goals and enactment history. Such a meta-model is suitable for modeling the process of requirements development.

The phrase, *dialog meta-model* was chosen, rather than the more common phrase *process model*, due to the specialized modeling of dialog processes and the use of meta-modeling. The dialog meta-model can be instantiated to aid the contextual needs of a development group. The dialog support system provides a user interface tailored to the instantiated dialog model. As stakeholders engage in the dialog, the dialog support system provides automated processing; for example, notifying stakeholders of interesting dialog events, such as a violation of a dialog goal.

A. Dialog Support System Components

There are four main components of the dialog support system.

- *Dialog Statement Model*
  Statements are added to the dialog by the people, or agents, involved in the dialog. In the dialog statement model, there are two important subtrees in the statement typology:
  - *Information*. A passive statement that adds new information to the dialog directly, or by reference to some external information source.
  - *Operation*. An active statement that adds new information derived through some computation based on the state of the dialog as captured in the dialog forum.
- *Dialog Forum*
  The dialog forum is a statement history that includes the statements *asserted* or *retracted* as part of the dialog. Forum statements are instances of statement types that are specified in the statement typology. The statement instances include values for attributes, as well as a belief interval indicating the time at which the statement was asserted, and if it was retracted, the time of retraction. Essentially, the forum is a log of statements that occurred during the dialog. Dialog events (i.e., assertion or retraction) may be initiated asynchronously by different stakeholders. To keep stakeholders aware of forum activities, the dialog system can notify stakeholders of new dialog events.
- *Dialog Protocol*
  A dialog protocol is a declarative prescription of “dialog rules”, indicating such things as the relative order of statements, as well as their content. In the dialog meta-model, a dialog protocol is represented by a hierarchy of dialog goals that specify desired forum properties. Examples of dialog protocols include: Roberts Rules of Order, and the software development lifecycle. Enforcement of the dialog protocol may be carried out through statement constraints that restrict the addition of statements to the dialog. Conversely, statements may be unrestricted, but operations can analyze the forum to determine the degree of compliance to a dialog protocol.
- *Dialog Monitor*
  A dialog Monitor is a predefined operation. After each dialog event, each believed monitor is automatically activated by the dialog system. A monitor itself specifies conditions under which another operation shall be automatically executed. For example, one predefined operation is GoalCheck. It determines those statements, if any, that fail to satisfy a dialog goal. Thus, a mon-

itor can specify that a specific instance of GoalCheck shall be activated after every dialog event to maintain a report on a goal's status. (Predefined operations, such as GoalCheck and Monitor are defined in the same manner as user defined operations.)

B. Dialog Meta-Model Overview

The main components of the dialog support system are defined in the dialog meta-model, as illustrated in Fig. 2. This meta-model defines the generic system classes. Before the system can be applied, classes of the meta-model must be specialized for a specific dialog context. For example, in the context of requirements development, a statement type, called Requirement, may be made an isA subclass of the Information statement class. Such specialized classes define a context specific *dialog model*. During the application of the dialog system, instances of Requirement may be asserted as part of a requirements dialog. These *dialog model instances* capture the content of the stakeholder dialog.

The dialog meta-model is illustrated in Fig. 2 as an entity relationship diagram. It shows how a dialog Forum consists of a set of dialog Statements. Each Statement instance must be an instance of the Operation or Information class, or an instance of their (user defined) subclasses.

The GoalCheck operation is a predefined subclass of Operation. An instance of GoalCheck specifies an instance of a Goal that is to be checked when it is activated. (A goal itself may be defined as an and/or tree of goals.) A Goal is checked by determining if its properties hold true within a Forum. Each Goal Property is a logical expression that refers to dialog model instances of a specific Forum. When specific statements can be identified as failing to hold for a goal, their failures are associated with the goal.

As illustrated in Fig. 2, Monitor is also a predefined subclass of Operation. It is automatically activated when the properties of its trigger hold for a specific Forum. Once activated, it may activate

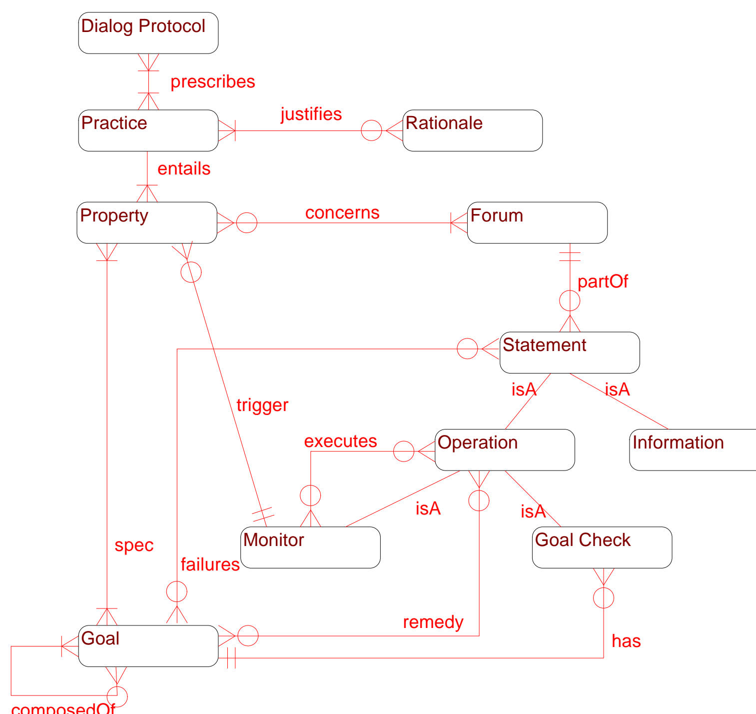


Fig 2. An entity relationship diagram of the main dialog meta-model entities.

other operations.

A Practice names a set of properties, and may be justified by Rationale. For example, defining user priorities for system requirements may be a practice of an incremental development protocol (i.e., a standard [41]) justified by their use in scheduling the incremental delivery of a system. The practice of defining user priorities for system requirements may be specified using logical properties of the Forum, as illustrated below.

$$\text{HAVEUSERPRIORITY} \equiv \forall R \in \text{REQUIREMENT}, \exists F \in \text{FORUM}, \exists P \in \text{USERPRIORITY} \bullet (\text{IN } R\ F) \wedge (\text{HASPRIORITY } R\ P)$$

(According to the definition of HAVEUSERPRIORITY, all Requirement statements must have an associated user priority.) A set of such Practices defines a Dialog Protocol.

While all entity instances of the dialog meta-model may be defined as logical expressions, only the properties of goals and monitors require it. In general, the entities of Dialog Protocol, Practice, and Rationale may be informal text. A dialog protocol is simply a name given to a set of practices. A dialog protocol is made operational by formally defining and composing dialog goals that represent the practices of the protocol.

C. Dialog Statements

Dialog statements are defined as standard object classes with constraints. The following provides a template for the definition of statements.

```
Class <statement-name> isA <Type> [, <Type>]* with
  attribute
     [ <attribute-name> : <Type> ; ]*
  constraint
     [ <constraint-name> : <constraint-expression> ; ]*
```

Statements types are organized into an inheritance (**isA**) typology. A user may attempt to add a statement to a forum by assigning attribute values to a statement type, thereby creating a statement instance. If the constraint expression does not evaluate to be false, then the statement is indeed asserted to the forum. (Constraints are logical expressions as illustrated in the following subsection.) Operational statements do more than simply assert attribute values. An assertion of an operational statement causes the execution of its associated (onAssert) method. Such methods may engage in computation and assign statement attribute values. Finally, the resulting statement is asserted to the forum.

D. Dialog Goals

As HAVEUSERPRIORITY illustrates above, dialog goals may be specified in terms of forum properties. Often a goal specification, like that of HAVEUSERPRIORITY, concerns properties of a certain statement type. In such cases, the dialog system can determine the specific statements that fail the goal. (Indicated as failures in Fig. 2.) However, a goal specification may concern summary properties across sets of statements, such as the number of high priority requirements:

$$\text{LessThan15HighPriorityRequirements} \equiv$$
$$\forall R \in \text{HIGHPRIORITYREQUIREMENT}, \exists F \in \text{FORUM}, \exists N \in \text{INTEGER} \bullet (\text{IN } R\ F) \wedge (\text{COUNT } R\ N) \wedge (<\ N\ 15)$$

The goal specification for LESSTHAN15HIGHPRIORITYREQUIREMENTS is met when there are less than 15 high priority requirements. However, there are no particular statements that will fail the goal specification. (One may consider the fifteenth high priority requirement asserted as the statement that caused the goal failure; however, this captures only part of the goal's intent.)

In general, a dialog goal expresses arbitrary logical formulas concerning a forum. A change in

any value within the scope of a formula can cause its evaluation to become false, thereby making failure attribution difficult. For example, a goal concerning a deadline may fail with the passage of time. Nevertheless, the dialog system can determine if such a goal specification is met in a forum. Moreover, specific statements that fail a goal can be determined for dialog goals that characterize properties of a specific statement type.

Goals also have an intentional mode that is used in combination with the goal specification. A goal mode may be *achieve* or *avoid*. The goal mode alters the way in which satisfaction of the goal specification is interpreted. In achieve mode, if the goal specification is met, then the goal succeeds. Conversely, in avoid mode, if the goal specification is met, the goal fails. (This leads to the corollary, *achieve*(g) = *avoid*(¬g).)

A third goal mode, *maintain*, is provided for convenience. A goal with mode *maintain* that is monitored prevents the assertion of any statement that violates the goal's specification. Thus, a single goal in *maintain* mode can be used in lieu of assertion constraints in multiple statement types. The statement *maintain*(g) is similar to *achieve*(g); however, if *maintain*(g) is Monitored, then statements cannot be asserted as the goal fails.

The two dimensions of goal mode and specification provide a simple means to alter the interpretation of goals. For example, in the early phase of a project, one may apply HAVEUSERPRIORITY as an *achieve* goal. During this phase, requirement statements may be asserted without user priorities—each such requirement causes the failure of HAVEUSERPRIORITY. Such failures typically lead to user notification and eventual statement modifications. During the final phase of the project, all requirements will have priorities. To prevent the addition of new requirements that do not have a user priority, the mode of HAVEUSERPRIORITY is set to *maintain*.

As illustrated in Fig. 2, a goal may also specify remedy operations. Such operations may be applied in the case of the goal's failure to reestablish the goal's satisfaction. GoalCheck is an operation that is used to activate goal remedies. When executed, it checks if the specified goal has failed. If so, and a Boolean runRemedy flag is set to true, then the remedies are executed.

Many goal failures may occur as the result of a dialog event. However, each remedy operation is executed in response to a single goal failure. Thus, there is no pre-defined global analysis of all goal failures followed by a global remedy. There are two means to address this problem.

First, a meta-goal concerning multi-goal failure can be asserted. For example, a slight modification of the LessThan15HighPriorityRequirements goal yields the goal MetaGoal-LessThan3ActiveFailedGoals.

MetaGoal-LessThan3ActiveFailedGoals ≡
∀ G ∈ ACTIVEFAILEDGOAL, ∃ F∈ FORUM, ∃ N∈ INTEGER • (IN G F) ∧ (COUNT G N) ∧ (< N 3)

The above goal fails if more than three active goals fail. The goal subclass ACTIVEFAILEDGOAL contains those goals that are currently being monitored and have failed. A remedy for MetaGoal-LessThan3ActiveFailedGoals can conduct global analysis of the failed goals in order to provide a more global remedy. In general, as an operational statement, a remedy operation may do complex programming activities, possibly involving user input, to modify the forum; for example, to modify all statements that led to a goal's failure.

A second means to remedy operations concerns composite goals. A composite goal specifies an and/or goal tree. A composite goal fails if one of its and-goals fails, or if all of its or-goals fail. When GoalCheck is applied to a composite goal, remedies at each level of the goal tree are applied to the failed subgoals.[1] Thus, the goal tree is used to: specify goals, recognize their failure, and apply remedies to bring about goal success (cf., [29]).

E. Dialog Protocols

Dialog goals are used to operationalize a dialog protocol. The practices of a dialog protocol are specified in the formal properties of dialog goals. Dialog goals themselves may be organized in an and/or goal tree. Thus, a dialog protocol is formalized as a composite dialog goal. A dialog protocol is activated (or inactivated) through monitoring operations. [2]

F. Monitoring

Monitor is a predefined operation, as illustrated in Fig. 2. A monitor has a trigger that specifies properties that must hold for the monitor to be active. A monitor also specifies operations that are to be executed when the monitor is active. After each dialog event, the operations of each active monitor are executed.

Monitoring can be used to automate two types of tasks. First, basic operations can be automatically run. This includes keeping analysis current and automating synthesis. Second, goals can be monitored to alert (or remedy) when goals fail.

Commonly, a monitor specifies that an operation shall be executed after every dialog event. For example, a specific operational statement, such as GoalCheck(HaveUserPriority), can be monitored to ensure that a goal failure of HaveUserPriority is immediately detected, and possibly remedied.[3] However, some operations are computationally expensive; for example, checking goal failure in a large goal tree or applying a complex remedy. In such cases, the monitor can be used to selectively invoke the operation. Monitors may be run periodically; for example, modulo the forum event count, or chronological time.

Monitoring may inadvertently introduce recursion among operations. After each dialog event, each monitor is activated. As a result, remedy operations may be executed and their results asserted as statements. The new statements may, in turn, activate more remedy operations that lead to more statements, etc. As a pathological example, consider a goal HaveResponse. It specifies that all statements must have a response.

$$\text{HAVERESPONSE} \equiv \forall\, S \in \text{STATEMENT}, \exists\, F \in \text{FORUM}, \exists\, R \in \text{STATEMENT} \bullet (\text{IN } S\ F) \wedge (\text{RESPONSE } S\ R)$$

A remedy operation for HaveResponse could add a new statement, S_r, as a response. Of course, the new statement, S_r, will need a response as well. In general, such recursion must be controlled through careful definition of goals and remedy operations. [4]

As a final point on monitoring, notice that monitoring of monitors falls naturally from the framework. For example, it can be specified that the goal HAVEUSERPRIORITY should be periodically checked by GoalCheck, as illustrated below:

$$\begin{aligned}\text{MONITORPRIORITY} \equiv\ & \exists\, M \in \text{MONITOR}, \exists\, F \in \text{FORUM}, \exists\, GC \in \text{GOALCHECK} \bullet \\ & (\text{IN } GC\ F) \wedge (GC\ \text{GOAL HAVEUSERPRIORITY}) \wedge \\ & (\text{IN } M\ F) \wedge (M\ \text{OPERATOR } GC) \wedge (M\ \text{EVENTPERIOD } 5)\end{aligned}$$

The goal MONITORPRIORITY is met when the goal HAVEUSERPRIORITY is checked periodically by a

[1] The current implementation follows a planning paradigm with remedies of the leaf goals executing first, followed by higher subgoals, ending with the topmost goal remedies.

[2] Defining a dialog protocol as a composite goal makes it possible to define multiple dialog protocols for a forum. Each dialog protocol may be activated (or inactivated) through monitoring operations on its composite dialog goal. While this feature has been useful in experimenting with dialog protocols, it has not been used in practice.

[3] The notation GoalCheck(HaveUserPriority) indicates that a specific operation, GoalCheck, is executed with the specified arguments, HaveUserPriority.

[4] The current implementation does provide some support for stopping recursion of an operation. It does not execute a new operation of type S, if the new operation is activated in response to another statement of type S, and that other statement was asserted in the last monitored cycle.

monitor, M. (The monitor, M, executes GoalCheck on the HAVEUSERPRIORITY goal.) Of course, since MONITORPRIORITY is a goal, it can also be monitored.

G. Hypothetical Statements

With statement constraints, goal specifications and their remedies, and operation monitoring, the consequences of adding a statement to the dialog can be difficult to predict. To facilitate user understanding, the dialog system provides an assertion mode for statements. In *standard* mode, a statement is asserted followed by the activation of monitors and their operations. In *hypothetical* mode, a statement is tentatively asserted and the subsequent events of monitors and operations are displayed but not asserted. Such a mode allows users to see the consequences of adding a statement to the dialog.[5]

III. TOOL SUPPORT FOR THE DIALOG META-MODEL

The dialog meta-model and support system form a conceptual design for the support of requirements analysis dialogs. To validate the effectiveness of the design, we constructed a software implementation. In our implementation environment, we aimed to support not only our current design, but the iterative refinement of our design as well.

Our dialog support system implementation is called DEALSCRIBE. It is part of our DEALMAKER tool suite aimed at supporting requirements negotiation[63].

A. Dialog System Architecture

The three main components of DEALSCRIBE interact over a network interface.[6]

- **Database Server**. The database server: stores dialog messages, checks constraints and rejects messages that violate them, answers queries of goal status and active monitors, and answers hypothetical queries. The deductive database, ConceptBase, provides these functions. It provides a concurrent multi-user access to O-Telos objects[26]. All classes, meta classes, instances, attributes, rules, constraints, and queries are uniformly represented as objects[49]. ConceptBase provides a powerful operational modeling language based on Datalog with negation[7][45]; it will terminate and produce the correct answer for any query it receives. Thus, one can formally describe a model in ConceptBase, populate it with instances, and have ConceptBase answer queries about the model or instances of it.
- **User Interface Server**. The user interface server provides: message input forms, dialog forum and message views, multiple forums, email notification of new messages, and secure administration of user access. A modified version of the World Wide Web discussion system, HyperNews, provides these functions. In our modified version, a user can post *typed* messages to forums, where the types are defined in the database server. A view of the forum can provide an overview of the discussion, where messages are laid out in an tree format that shows replies to a message indented under it (see figure 6). Statements are stored in ConceptBase and as WWW pages. Thus, one can run both ConceptBase (logical) queries and WWW search engine (match-based) queries, such as provided by Excite. Finally, WWW collaborative tools (e.g., whiteboard) can be invoked from DEALSCRIBE, as an operation, and its content stored into a forum.

[5] One could argue that such hypothetical considerations should be made part of the dialog history. An anonymous reviewer remarked that this could be done by managing a history tree. The current implementation uses a linear statement history with hypothetical statements because of: the smaller history it entails, the simplicity of the user interface, and the ease of implementation.
[6] All components communicate using POSIX compliant TCP connections.

- **Clients**. A WWW browser serves as the basic user client. Using this network interface, multiple clients can interact with DEALSCRIBE from any internet connection. Additional client interfaces include a graphical browser and a (Perl) program API.

In the following subsections, we summarize how the dialog system functions are implemented.

B. Dialog Statements

The statement template of section II.C is essentially the syntax of ConceptBase class definitions. Thus, a DEALSCRIBE information statement, such as Requirement, can be defined similarly. For example, Requirement with attributes perspective, mode, description, and contention can be defined as follows:

```
Class Requirement isA InformationStatement with
  attribute
    perspective : Perspective;
    mode : Mode;
    description : String;
    contention : FuzzyNumber
end
```

From such definitions, DEALSCRIBE generates an input form. A user can fill in, or select, values for attributes of the statement object. Operation statements are similarly defined. For example, attributes of GoalCheck are defined as follows:

```
Class GoalCheck isA OperationStatement with
  attribute
    goal : DialogGoal;
    runRemedy : Boolean;
    resultString: String;
    result : Proposition
end
```

Like information statements, the object attributes of operation statements may serve as input fields; similarly, others may serve as output fields. All operation statements have an associated method that is executed. (Currently, all methods are written in the Perl programming language.)

Figure 3 shows a portion of a GoalCheck statement from DEALSCRIBE. As a result of the user selecting a goal and then posting the operation, GoalCheck found statements that failed the goal and presented them as WWW links.

Figure 4 shows a portion of a DEALSCRIBE WWW page for adding a statement. This table of statement types, and their associated input and output forms, are automatically generated from the corresponding ConceptBase statement model. The ConceptBase statement hierarchy is depicted in the graphic browser of figure 5. A user can post a statement by selecting a statement type and then the “Add” button. All such aspects of the user interface that are dependent on the meta-model are automatically generated. Thus, modifying a dialog model, even dynamically, is easy. Simply define a new statement type and the statement buttons and I/O forms are automatically updated.

C. ConceptBase Statement and Query Semantics

ConceptBase classes, like the above GoalCheck, define data types that include typed attributes and constraints. These data types, and their instances, are stored in a deductive database. As GoalCheck illustrates, ConceptBase is an object-oriented database. Moreover, it uniformly integrates concepts of deductive databases.

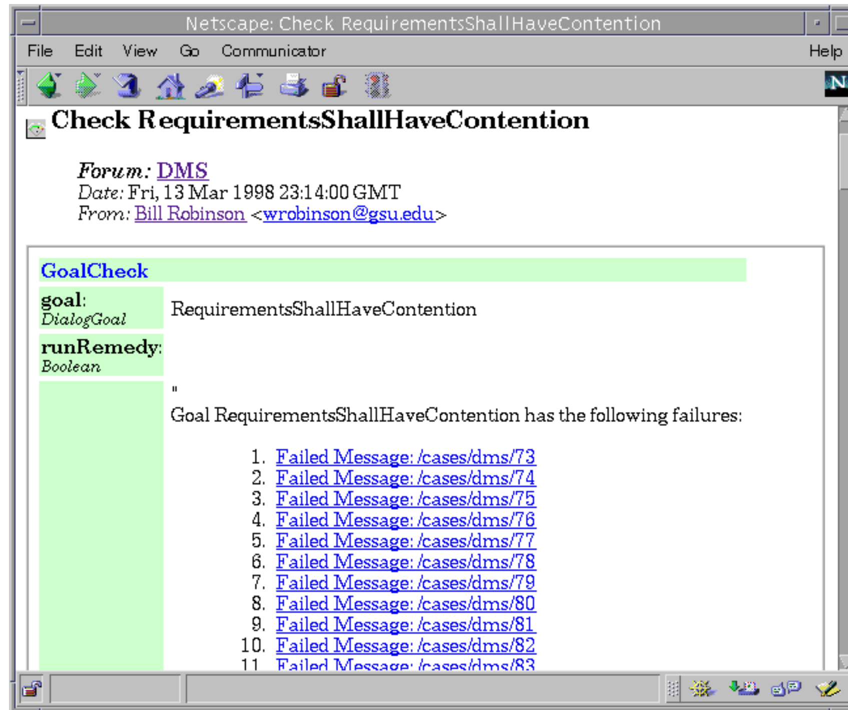


Fig 3. A portion of a DEALSCRIBE WWW page showing the results of asserting a GoalCheck statement. The page is representative of the input and output forms. Each object attribute is depicted as a row heading in a table. Attribute values are depicted as the contents of table cells.

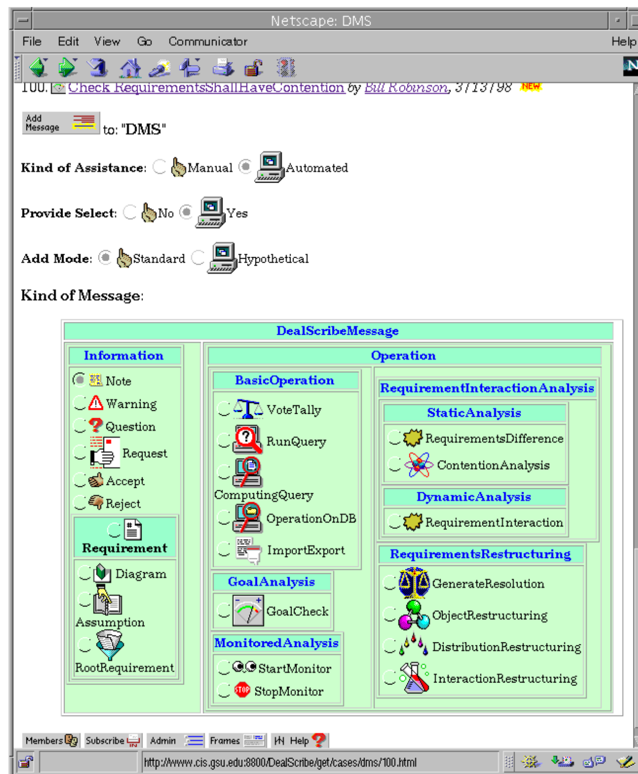


Fig 4. A portion of a DEALSCRIBE WWW page showing the selection of statement types as a hierarchy of radio button tables. Each type is defined in the ConceptBase statement model.

> Without leaving the Datalog (with negation) framework, it makes object-oriented abstraction mechanisms available to the user, thus providing significant help for data structuring as compared with relational deductive databases. ... The query language supported is similar to those offered by other object-oriented databases but, like in deductive databases, more directly integrated with the rest of the data model; this has led to some useful ideas with many applications, such as the parameterization of query classes.[27]

ConceptBase provides deductive databases techniques, such as such arbitrary relationships between tables, recursive processing of tables, and parameterized query classes. While such techniques are being incorporated in commercial databases, most commercial systems still lack these features[45]. Nevertheless, these features have been helpful in development of DEALSCRIBE—particularly, the parameterized query class.

ConceptBase query classes are defined like its data classes. A QueryClass is itself an instance of one or more classes. Through its constraint and **isA** specification, a QueryClass defines necessary and sufficient conditions for objects that are instances of it. Such conditions are used to compute the objects that answer the query.

Consider the following HaveUserPriority QueryClass.

```
QueryClass HaveUserPriority in DialogGoal isA Requirement with
mode
    mode_a : Achieve
constraint
    ID_Priority : $ exists x/Priority (this userPriority x) $
end
```

It is an instance of the DialogGoal class. Objects that are members of (i.e., **in**) HaveUserPriority are constrained to be **isA** the Requirement class and have a value for attribute named userPriority.[7] Thus, query definitions may themselves be instances of one class, while having objects fulfilling their query constraints from another class. (This is not unlike a standard SQL query; however, typically, a SQL query's type cannot be queried.)

QueryClass definitions can be queried and manipulated in ConceptBase. Such uniformity of manipulation provides a concise way to define and check dialog goals. For example, one can define an operation that queries goals of the current dialog protocol as part of proving interesting properties, such as their logical consistency. However, such meta-goal analysis, beyond goal violation checking, is not part of DEALSCRIBE.

D. Dialog Goals

The DialogGoal class is defined similar to statement types, as shown below:

```
Class DialogGoal  with
 attribute
    mode : CheckMode;
    remedy : DealScribeOperation;
    andGoals : DialogGoal;
    orGoals : DialogGoal
 end
```

As illustrated above, a specific goal is defined as an instance of this type.

Goal failure can also be determined through a database query. Consider the achievement of the HaveUserPriority goal. If a Requirement statement does not have a userPriority, then the goal fails. More generally, if a goal mode is Achieve, then statements in the **isA** class specified that do not sat-

[7] In the ConceptBase query notation, "this refers to the instance retrieved from the database—in HaveUserPriority, a Requirement. Other ConceptBase notations include: x/Type, which constrains variable x to be an instance of class Type, and (this userPriority x), which constrains the value x of attribute userPriority for object this.

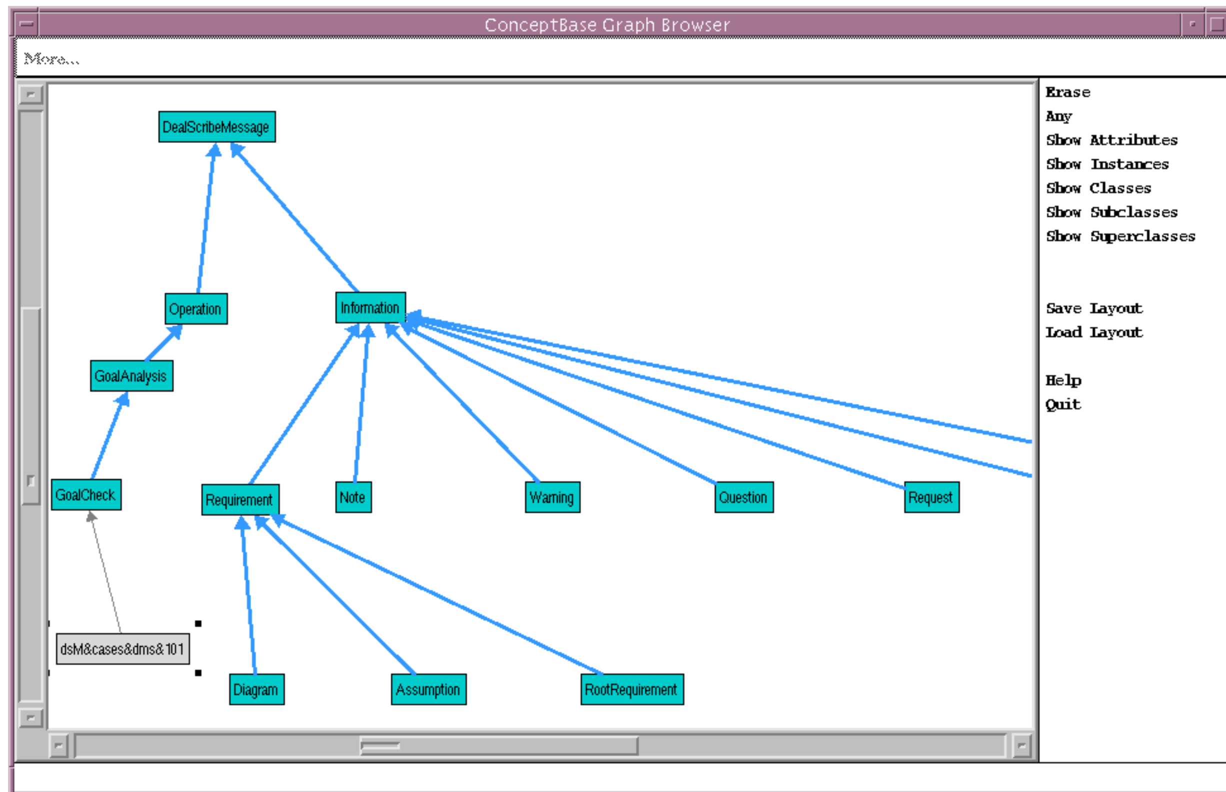


Fig 5. A portion of the ConceptBase database showing part of the hierarchal statement model (as viewed from ConceptBase's graphical browser).

isfy the constraint are statements that fail the goal. This is captured in the following rule:

Failure Retrieval Rule::
For a goal *G* with isA type *T* and with constraint *C*, goal failures are in type *T* that satisfy *not C*.

Since a DEALSCRIBE goal is itself a query, it can be used to find those statements that are instances of the goal type, but are not instances of the goal query. (If a goal mode is Avoid, then it finds statements that do satisfy the goal query.) The following query implements goal failure detection for dialog statements.

```
GenericQueryClass FailuresOfGoal isA DealScribeMessage with
parameter
   goal : DialogGoal;
   mode : CheckMode
 constraint
   c : $ ((mode == Avoid) and
         (exists parent1/OneLinkIsA[goal/child]  (this in parent1)
                and not (this in goal)))
        or
         ((mode == Achieve) and
          (exists parent2/OneLinkIsA[goal/child] (this in parent2)
                and (this in goal)))  $
end
```

FailuresOfGoal is a parameterized query, which given a goal and a mode, will return those statements that satisfy the constraint. The FailuresOfGoal's constraint in turn implements the Failure Retrieval Rule above. (OneLinkIsA returns the direct parent **isA** types of the goal.)

Goals that concern summary properties of statements are analyzed separately. For example, LessThan15HighPriorityRequirements is a instance of a CountingGoal. Its attributes include: a query that is used to accumulate the items to be counted, a goal_count indicating a value to be compared

using the indicated relation.

```
QueryClass LessThan15HighPriorityRequirements in CountingGoal with
mode
   mode_a : Achieve
query
   q : HighPriorityRequirement
goal_count
   gc : 15
relation
   c : LessThan
end
```

CountingGoal is a subtype of ComputingGoal. These goal types require some (even simple) algorithmic computation that cannot be expressed in ConceptBase queries. To accommodate such goals, DEALSCRIBE computes such values externally and caches them with the ConceptBase goal. Such caching is efficient, as only those goals that are currently active will be updated.

In general, failure for a goal tree is implemented with a recursive query that descends the and/or goal tree. This query, FailsGoalTree, uses a slightly extended version of the above FailuresOfGoal to incorporate computed goals. When applied as part of the GoalCheck, the user is presented a goal tree that includes WWW links to the messages that fail each goal. (See Fig. 4) If the user chooses to apply the associated goal remedies, they will be applied, in order, from the leaves to the root of the goal tree.

E. Monitoring

Monitors are defined like all other statement types. The following ConceptBase class defines StartMonitor that is used to define a monitor instance. (A similar statement is used to stop a monitor.)

```
Class StartMonitor isA MonitoredAnalysis with
 attribute
   Trigger : Query;
   StartTime : DateTime;
   EndTime : DateTime;
   TransactionInterval : Integer;
   TimeInterval : Integer;
   Operator : DealScribeOperation
end
```

To use StartMonitor, a user selects an operation to be monitored and a trigger, that when non-nil, will result in the execution of the operation. While a trigger may involve complex temporal expressions, most triggers simply define start and end times or transaction intervals. For convenience, these may be input directly.

Monitored operations are executed after every dialog event. To do so, the ActiveMonitors query retrieves instances of StartMonitor whose triggers are satisfied. Next, DEALSCRIBE executes each associated operation and a record of its operation is asserted into the dialog forum. As with all statements, an email message will also be sent to users that have indicated that they desire email notification.

Any operation statement can be monitored. To do so, 1) a user asserts an operation statement, O, then 2) a user asserts a StartMonitor statement as a response to O. The original assertion of O allows for the input parameters of O to be assigned. The assertion of the StartMonitor defines the conditions under which operation O will be executed. DEALSCRIBE will run the operation, according to the monitor parameters, until a StopMonitor is asserted for O. The forum, as depicted by DEALSCRIBE in figure 6, indicates: 1) the initial assertion of GoalCheck, 2) the subsequent StartMonitor, 3) the

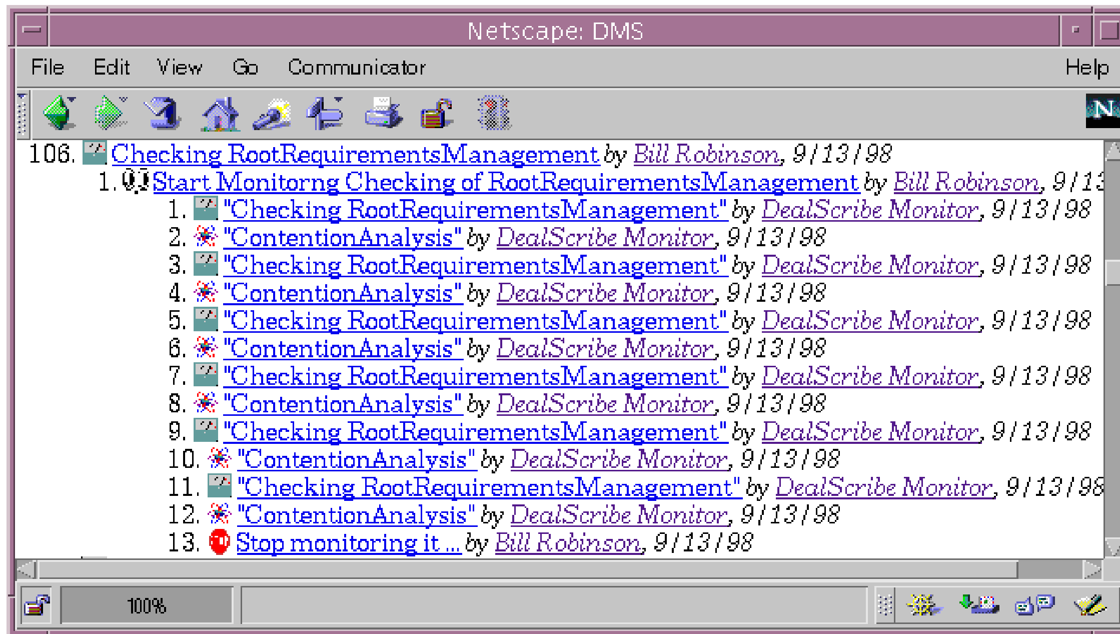


Fig 6. A portion of a DEALSCRIBE WWW page showing statement headings: <*number, icon, title, author date*>. The initial GoalCheck statement is at the top, followed by a StartMonitor response, and the subsequent monitored responses of three types: GoalCheck, ContentionAnalysis, and StopMonitor. Note that ContentionAnalysis is remedy of a subgoal of the monitored RootRequirementsManagement goal. The final StopMonitor response ends this monitoring of GoalCheck. (Responses are shown indented, below, and with the newer statements toward the bottom.)

subsequent execution of the monitored GoalCheck, and finally, 4) the StopMonitor statement. Thus, monitoring is divided into two parts: 1) the condition under which the operation will be executed, and 2) the operation itself. Additionally, the operation may have its own conditions that must be met before results are asserted.

Figure 6 shows a portion of a DEALSCRIBE forum page. The initial assertion of the GoalCheck(RootRequirementsManagement) operation is shown at the top of the WWW page, while the StartMonitor, and subsequent responses, are shown below. As a result of applying Check(RootRequirementsManagement) early in the requirements development process, a remedy in the RootRequirementsManagement tree, ContentionAnalysis, was automatically applied. Thus, both the monitored operation, Check(RootRequirementsManagement), and its applicable remedies are executed as the result of monitoring.

F. Hypothetical Statements

While the dialog meta-model provides for the expression and monitoring of complex interactions, the consequences of asserting any one statement can be difficult to determine. Of course, consequences can be determined by simply asserting a statement. However, that may lead to a number of unwanted warning messages or remedy operations. In cases where the user would like to explore hypothetical statements, DEALSCRIBE does not assert a statement, but rather tentatively asserts the statement and then queries about the resulting state.

DEALSCRIBE uses TELL-HYPO to hypothetically run one DEALSCRIBE dialog event. In standard mode, DEALSCRIBE executes the ConceptBase TELL function to add statements to the database. However, in hypothetical mode, DEALSCRIBE executes the ConceptBase TELL-HYPO function to

temporarily add statements to the database, check constraints, and answers queries. Thus, DEALSCRIBE hypothetically asserts the statement, queries to find the hypothetical ActiveMonitors, and determines the consequences of the monitored operations.

Consider the standard addition of a requirement without an assigned user priority. If such a property were being monitored, say through GoalCheck(HaveUserPriority), then a warning of goal failure would be asserted. Similarly, in hypothetically asserting the requirement, the same goal failure would be noted, but no actual warning message would be asserted.

G. Related Systems

The DEALSCRIBE implementation simply demonstrates one means to support the functionality of the dialog meta-model and system. We considered other support environments (e.g., DOORS[54], RTM[39], RDD-100[1]), but selected to create DEALSCRIBE instead. Adapting a commercial tool to support the dialog meta-model may have created a more usable system. However, our goal was to develop an environment for iterative development and experimentation of dialog meta-models and supporting tools. Consequently, DEALSCRIBE was built on a meta-modeling tool that provided flexibility in prototyping.

Current requirements engineering tools do not have process model support for the goal tree and monitor specified in the dialog meta-model and system. However, many do have a means of adding tool extensions (i.e., an API) by which process support can be provided. For example, a similar model of standard compliance checking was built upon DOORS[18].

This exploratory research required a flexible meta-modeling environment. Most requirements engineering tools use a relational database to support their analyses. While such standardization has advantages, the traditional relational database implementation makes meta-modeling difficult. In a relational database, queries apply to data instances (records) but not to data schema. (Such schema form the meta-model of Fig. 2.) In contrast, both schemas and their instances are stored and queried in ConceptBase.

The meta-modeling environment of ConceptBase aided the iterative and experimental development of the dialog meta-model and support system. By querying the meta-model, external tools could adapt themselves to changes in the meta-model. For example, DEALSCRIBE input and output forms were automatically updated with changes to the meta-model of Fig. 2. QueryClasses also provided for analyses of model instances that were themselves schemas. For example, dialog goals are themselves queries; yet, queries such as FailuresOfGoal, can be formulated to analyze dialog goals. These meta-modeling features have simplified the development of DEALSCRIBE. While such complexity may not be necessary for a commercial requirements tool, it has aided the development and experimentation of the dialog meta-model.

Few commercial requirements engineering tools have a language or support environment based on typed predicate logic expressions. Most queries and constraints are based on SQL or simple object model constraints. In contrast, ConceptBase provides an efficient implementation of Datalog with negation. This language simplifies the transition from the formal theoretical model to the running implementation, as illustrated from sections II to III.

Some research environments may be adapted to support the dialog meta-model and system. The most obvious is the DOORS requirements tool extended to support compliance management[18]. The extended environment can monitor the compliance of a requirements document to standard documentation *practices*. Each practice references document properties as represented in the underlying DOORS database. Then, an external monitor, FLEA[20], raises notifications when a

practice is not met. DEALSCRIBE has similar capabilities; however, differences include: 1) DEALSCRIBE's goal language is based on the logically expressive Datalog[45], 2) DEALSCRIBE goals can refer to operation properties as well as document properties, 3) DEALSCRIBE explicitly includes remedy operations that can reestablish goals, 4) DEALSCRIBE provides hypothetical statements, and 5) DEALSCRIBE is implemented in a meta-model environment, thereby facilitating experimentation and extension.

WinWin is a commonly cited research project on requirements development groupware[5][17]. The WinWin tool provides groupware support for tracking team development of requirements, including conflict detection and resolution. WinWin notifies analysts when new requirements may conflict with other requirements, according to its hierarchy of requirements conflict criteria. In this sense, it provides monitoring and notification. However, the criteria for notification is encoded into system procedures, thereby making formalism, dynamic criteria changes, or dynamic disabling and enabling of monitors difficult. Finally, the groupware aspect of the system is based on a paradigm of message exchanges, as opposed to a globally viewable message forum.

IV. DEVELOPMENT GOALS FOR MANAGING INCONSISTENCY

A dialog support system, like DEALSCRIBE, can be applied to a variety of tasks that support the management of a requirements dialog. In general, development goals that can be expressed in terms of logical expressions over informational and operational dialog statements can be monitored and maintained. For example, DEALSCRIBE goals can represent aspects of standardized processes, such as ISO 9000 or IEEE process descriptions[41], and can execute warnings in response to goal failures (cf., [18]). In addition to standards compliance warnings, DEALSCRIBE can take an active role in executing tasks. Goals can specify remedy operations that can be applied in the case of goal failure. Such goals, in combination with monitoring, can automate tasks such as: synthesizing resolutions as requirements conflicts arise, totaling of votes for alternative requirements at the end of a voting period, or updating the status of requirements with the changes in belief of dependent assumptions (cf., [56]). In these ways, DEALSCRIBE supports the asynchronous work of analysts through monitoring and maintaining of development goals.

This section demonstrates how development goal monitoring can take an active role in managing requirements development. Rather than continue with the HaveUserPriority example, a requirements conflict detection and resolution dialog protocol, called Root Requirements Management, is introduced. This protocol specifies dialog goals that support its particular form of requirements conflict management. While this protocol is not an accepted standard, it does specify goals concerning analyst interactions (e.g., voting), as well as goal remedy operations. As such, Root Requirements Management will more fully demonstrate how development goal monitoring can take an active role in managing requirements development.

A. Root Requirements Management

We have developed Root Requirements Management as a dialog protocol for managing requirements conflicts. It is aimed at addressing two basic objectives: 1) conflict understanding and 2) conflict removal. First, to aid conflict understanding, we have developed Root Requirements Analysis[64]. This technique uncovers requirements conflicts, analyzes them as a group, and directs analysis toward key conflicts. Second, to generate alternative resolutions for each conflict, we have developed Conflict-Oriented Requirements Restructuring (CORR)[62]. To demonstrate how development goal monitoring can apply to more complex protocols, we show how a Root Require-

ments Management protocol can be defined in terms of goal monitoring. The protocol consists of: 1) Root Requirements Analysis, 2) Conflict-Oriented Requirements Restructuring, and 3) resolution selection.

Root Requirements Analysis

Two objectives of Root Requirements Analysis are: (1) to understand the relationships among the requirements, and (2) to order requirements by their degree of contention. This information can be used to guide other analyses, such as conflict resolution through Conflict-Oriented Requirements Restructuring.

The overall procedure of Root Requirements Analysis is:

1) **Identify root requirements** that cover all requirements in the requirement document. Requirements are generalized to derive *root requirements*,
2) **Identify interactions** among root requirements. Root requirements are pairwise subjectively compared to derive root requirements interactions; the interaction types are: Very Conflicting, Conflicting, Neutral, Supporting, and Very Supporting
3) **Analyze the root requirement interactions**. Requirements metrics are derived from the root requirements interactions.

This technique is important in that it provides a systematic method by which requirements conflicts can be surfaced and then systematically selected for efficient resolution.

Root Requirements Analysis Metrics

Once the root requirements conflicts are identified, they can be used to derive useful metrics. Three that are particularly helpful are: relationship count, requirement contention, and average potential conflict. *Relationship count* is simply a count, for all root requirements, of the number of interactions a root requirement has with other root requirements, for each of the five types of relationships. *Requirement contention* is the percentage of all relationships the requirement participates in which are conflicting; thus, if a requirement's contention is 1, then it conflicts with every other requirement in the requirements document. Finally, *average potential conflict* is the conflict potential of a requirement averaged across all of its conflicting relationships.

We have found these metrics to be useful in ordering the resolution of conflicts. For example, we have found that resolving the most contentious requirement first not only directly resolves one conflict, but often it indirectly resolves others[64]. Thus, resolving high contention requirements first is a practice of Root Requirements Management that is supported by a dialog goal (ResolveHighestContensionFirst in §IV.B).

Resolution Generation through Requirements Restructuring

The purpose of resolution generation is to remove conflict by finding substitute requirements that fulfill the intent of the original requirements, but without their undesired consequences. Resolutions can be generated by altering the structure of the original requirements through transformations. Resolutions fall into three general classes: (1) selection among the original conflicting requirements, (2) selection among restructurings of the conflicting requirements, and (3) selection of run-time monitoring and recovery of the original conflicting requirements[60]. Such resolutions are generated through a combination of two classes of restructuring transformations:

1) *Object restructuring* transformations, that use object relationships (e.g., **isA** and **partOf**) to find related objects for substitution.
2) *Condition restructuring* transformations, that distribute conflicting positions across specific contexts, thereby conditionalizing the state under which each requirement will be met.

These transformations are applied as a part of the practice of Root Requirements Management. As such, they are supported as remedies of the dialog goal, ResolveHighestContentionFirst.

Resolution Selection

Once resolutions are generated for each conflict, the analysts must select which resolution(s) will become part of the next requirements baseline. Voting is an efficient means for group selection. Resolution selection through voting is a subgoal of the Root Requirements Management protocol.

B. A Dialog Protocol for Root Requirements Management

The following summarizes the dialog practices of Root Requirements Management. Such practices can be translated into DEALSCRIBE dialog goals as part of the protocol for Root Requirements Management.

1) RootRequirementsAnalysis
   The analysis portion has three basic subgoals:
   (a) DeriveRoots
      No more than 3 percent of all requirements shall lack an associated root requirement.[8]
   (b) DeriveInteractions
      No more than 5 percent of all root requirements shall lack an associated description of conflict, called a requirement interaction.
   (c) DeriveContention
      No more than 5 new interactions shall exist before requirements contention analysis is conducted.
2) GenerateResolutions
   Resolutions shall be generated for each requirements conflict according to the following goal:
   (a) ResolveHighestContentionFirst
      Resolutions shall be generated for conflicts among the most contentious requirements. To generate resolutions, do the subgoals of generating: object, distribution, and interaction resolutions.
3) SelectResolutions
   Resolutions shall be selected for each requirements conflict according to the following goal:
   (a) VoteOnResolutions
      A vote shall be conducted where there are multiple resolutions of requirements conflict.

Each of the above practices is translated into a DEALSCRIBE goal in the following subsections.

Root Requirements Dialog Goals

The first goal, DeriveRoots, simply states that no more than 3 percent of all requirements shall lack an associated root requirement. The definition makes use of RequirementsWithoutARoot, which indicates those requirements that have not been analyzed for roots. DEALSCRIBE can determine the percentage of requirements (query2) that lack an associated root requirement (query1) in the fol-

[8] Such percentages are simply intended to illustrate numeric goals. Percentages in goals determine how much deviation from compliance a dialog may have. Once failure is reached, DEALSCRIBE provides a warning (and possibly a remedy) after each dialog event until there is no goal failure. In a circumstance where an analyst is required to manually remedy a goal failure, a percentage goal allows an analyst some time prior to automated warnings. Of course, an analyst can directly apply GoalCheck to a non-percentage goal (e.g., RequirementsWithoutARoot) to immediately determine any failures.

lowing definitions.

```
QueryClass RequirementsWithoutARoot  isA Requirement with
 constraint
   NoRoot : $ not exists r/RootRequirement (r requirements this) $
end

QueryClass DeriveRoots in PercentGoal with
mode
    mode_a : Achieve
query1
    q1 : RequirementsWithNoRoot
query2
    q2 : Requirement
goal_count
    gc : 3
relation
    c : LessThan
end
```

The goal, DeriveInteractions, is identical to DeriveRoots except that query1 retrieves RootsWithoutAnInteraction. Also, the DeriveContention goal is similar but uses counting instead of percentage. However, there is one additional attribute that is part of the goal. DeriveContention's remedy attribute indicates an operation statement that should be executed automatically if goal failure occurs. Upon goal failure, a remedy operation is passed the dialog goal and results of the goal-checking query. In this case, upon failure of DeriveContention, ContentionAnalysis is executed and the results are asserted to the dialog.

The composite goal for Root Requirements Analysis is the conjunction of the above three goals, as shown below.

```
QueryClass RootRequirementsAnalysis in DialogGoal with
mode
    mode_a : Achieve
andGoals
    g1 : DeriveRoots;
    g2 : DeriveInteractions;
    g3 : DeriveContention
end
```

Resolution Generation Dialog Goals

The resolution generation goal is a bit more complex. Recall that DeriveContention associates a degree of inconsistency, called *contention*. The goal, ResolveHighestContentionFirst, seeks to resolve interactions among the most contentious requirements first. The goal can be defined as follows.

```
Class MostContentiousUnresolvedRequirements isA Requirement with
constraint
   ConReq : $ not exists gr1/GenerateResolution (gr1 requirements this) and
                      exists thisCon/Integer (this Contention thisCon) and
                      not exists otherReq/Requirement otherCon/Integer
                           ((otherReq Contention otherCon) and
                            (otherCon > thisCon) and
                            not exists gr2/GenerateResolution (gr2 requirements otherReq)) $
end
```

```
QueryClass ResolveHighestContentionFirst in DialogGoal isA RequirementInteraction with
mode
   mode_a : Achieve
remedy
   r1 : ObjectRestructuring;
   r2 : DistributionRestructuring;
   r3 : InteractionRestructuring
constraint
   RHF : $ exists req1,req2/MostContentiousUnresolvedRequirements
                  ((this requirements r1) and (this requirements r2)) $
end
```

The above definition of ResolveHighestContentionFirst makes use of a derived class, MostContentiousUnresolvedRequirements. This class is defined to be those requirements: 1) for which there has not been a resolution generated, and 2) there does not exist another requirement with a higher contention for which there has not been a resolution generated. Once MostContentiousUnresolvedRequirements is defined, specifying the goal ResolveHighestContentionFirst is easy. It is simply those requirements that are both: 1) in the MostContentiousUnresolvedRequirements, and 2) interact with each other, as denoted by both being in the requirements of the same RequirementInteraction. Thus, ResolveHighestContentionFirst makes use of the dialog forum to specify the goal of always selecting unresolved interactions among requirements with the highest contention.

The dialog goal, ResolveHighestContentionFirst, specifies three remedies that correspond to the three types of resolution generation methods described above in section IV.A. When ResolveHighestContentionFirst is monitored and fails, the three operations will be executed.

Resolution Selection Dialog Goals

The final goal (not shown) simply specifies that, where there is more than one resolution of a requirements conflict, there should be a vote. The goal, VoteForResolution, specifies this as a counting goal. Whenever the count of undecided resolutions is greater than one, the goal fails and the remedy of a VoteTally is executed. The operation, VoteTally, simply asserts a statement indicating the number of votes each resolution received. Of course, voting may be more complex, involving time periods and multiple rounds; however, VoteForResolution gives a flavor of how resolution selection can be included in the protocol.

Analysis and Resolution Dialog Protocol

The final protocol is represented as a composite goal that contains the above goals, as illustrated below.

```
QueryClass RootRequirementsManagement in DialogGoal with
mode
   mode_a : Achieve
andGoals
   g1 : RootRequirementsAnalysis;
   g2 : ResolveHighestContentionFirst;
   g3 : VoteForResolution
end
```

Once so defined, this goal model may be selected as part of the input to run the GoalCheck operation. The operation, GoalCheck(RootRequirementsManagement), can be monitored to ensure updated reports and remedies concerning Root Requirements Management. Of course, RootRequirementsManagement is but one goal model. Multiple goal models can co-exist simultaneously, or at different times. However, it is the responsibility of the user to ensure that multiple goal models do not defeat each other's goals, or enter into accidental loops through the interaction of remedy opera-

tions.

Summary

The proceeding definition of the Root Requirements Management protocol demonstrates how an interesting dialog protocol can be defined in terms of the dialog meta-model of sections II and III. The following section demonstrates that such a protocol definition can be effectively monitored as part of a requirements development dialog.

V. An Exploratory Case Study

An exploratory case study has been conducted to determine if dialog goals, as defined in section II, can be effectively monitored using the DealScribe implementation of section III. To do so, we activated the Root Requirements Management protocol in DealScribe and initiated a requirements development dialog. Consequently, we were able to determine that development goals, as implemented in DealScribe, can effectively be monitored.

As a secondary consideration, we were interested in exploring how development goal monitoring assists analysts in managing requirements. To facilitate such an assessment, we conducted a comparative study. First, the authors applied Root Requirements Analysis to a standard problem using only a spreadsheet and word processor[62][64]. Second, the authors again applied Root Requirements Analysis to the same problem, but with the support of DealScribe's development goal monitoring. Unfortunately, such a study lacks appropriate controls to be considered an experiment. However, as a comparative case study, it did give us a qualitative understanding of the utility of development goal monitoring.

In this section, we present an overview of the case studies and compare the manual application of Root Requirements Analysis with an assisted application.

A. Manual Analysis of the Requirements

To assess the utility of development goal monitoring, we applied it to an established requirements engineering problem, that of the distributed meeting scheduler software[71]. Figure 7 illustrates the result of applying Root Requirements Analysis. The initial 53 requirements led to dialog that uncovered 30 root requirements and 72 requirements conflicts[64]. Subsequent application of conflict resolution to 22 of the conflicts generated 150 resolutions[62].

Both analyses were incomplete in that all requirements were not completely analyzed for all conflicts or resolutions. Potts *et. al.* estimates 4.7 person-months for an Inquiry Cycle analysis of the requirements[53]. Based on our analysis at 1.2 hours per root requirement, a complete Root Requirements Analysis may add 0.5 person-months to the Inquiry Cycle analysis[64]. Based on our analysis at 1 hour per conflict, our resolution generation procedure may add 1 person-month.

B. Assisted Analysis of the Requirements

To assess the utility of DealScribe's development goal monitoring, we applied the Root Requirements Management protocol to the distributed meeting scheduler[71]. Since the requirements, root requirements, and their interactions were defined in the previous case study, their files were imported into DealScribe. Next, the Root Requirements Management protocol was activated. Consequently, DealScribe identified that the DeriveContention subgoal of RootRequirementsManagement had failed. It then applied the ContentionAnalysis operation as a remedy. Next, DealScribe identified that the ResolveHighestContentionFirst subgoal of RootRequirementsManage-

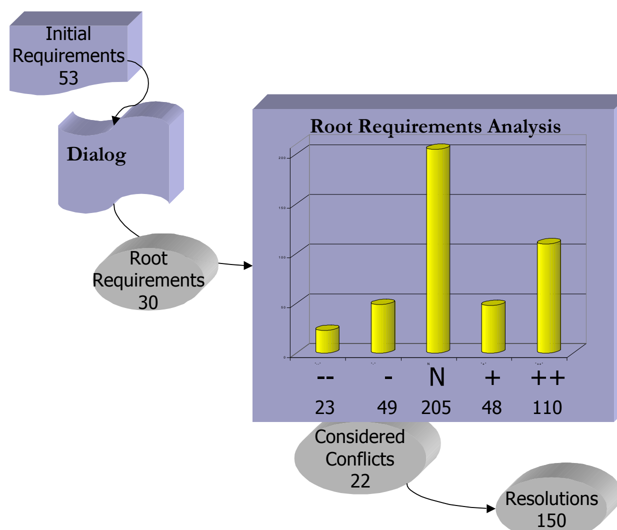


Fig 7. A depiction of the analysis of the distributed meeting scheduler showing how 53 initial requirements led to 30 root requirements, and then 72 conflicts. From the 22 conflicts considered, 150 requirement resolutions were generated.

ment had failed. Consequently, it applied resolution generation operations as a remedy. Next, DEALSCRIBE identified that the VoteForResolution subgoal of RootRequirementsManagement had failed. Consequently, after we voted for a resolution, DEALSCRIBE applied VoteTally as a remedy. Some further analysis was applied to extend the "manual" study. DEALSCRIBE interjected analyses as called for in the RootRequirementsManagement protocol.

Through the RootRequirementsManagement protocol, DEALSCRIBE automated two types of tasks. First, simple calculations, such as ContentionAnalysis, were updated as requirement interaction types were entered by analysts. Such calculations were also automatically updated by the spreadsheet in the "manual" case study[64]. Second, complex calculations were updated. These included the selection of the most contentious unresolved requirement and the generation of conflict resolutions, neither of which were supported in the "manual" case study.

The operations automated in DEALSCRIBE executed in a few seconds to nearly three minutes, depending on the number and complexity of the database queries.[9] Root requirements analysis was essentially the same in both case studies, as root identification and interaction detection were manual operations in both case studies. However, resolution generation was significantly improved, as many resolutions were automatically generated.[10] Roughly, one-half the effort of resolution generation was removed through the automated application of the generation procedure.[11] This provides a rough estimate of 1 1/6 months for the time to complete Root Requirements Management using DEALSCRIBE. This that compares favorably to the manually estimate of 1.5 person-months. However, this is a result of operation automation, and is not directly attributed to development goal monitoring.

C. Observations on the use of DEALSCRIBE

We believe the benefit of dialog goal monitoring rises as confusion about the dialog increases.

[9] ConceptBase version 4.1 on a moderately loaded Sparc 1000 running Solaris 2.6.
[10] DEALSCRIBE applied a static conflict restructuring strategy described in [62]. Additionally, we added our own resolutions.
[11] Properly structuring requirements in preparation for automation accounted for about one-half the effort in resolution generation.

As the number of requirements, analysts, and their analyses grows, confusion over the current state of requirements can grow.

Consider table I. It presents a snapshot of the DEALSCRIBE requirements dialog. It lists the number and type of each statement asserted by either DEALSCRIBE or the analysts; "other" statements were miscellaneous statements such as votes, monitor start/stop, or analyst's notes.[12] As would be expected, DEALSCRIBE asserted statements that were generated from the Root Requirements Management protocol or were in response to an analyst's request. Notice that even a partial analysis of the requirements results in a relatively large number of statements. As the number of statements grows, it becomes more difficult to determine if a development protocol is being met.

TABLE I DISTRIBUTION OF DIALOG STATEMENTS

	Number Asserted by:	
Statement Type	Analysts	DEALSCRIBE[A]
Requirement	1	53
Resolution	2	171
Root Requirement	1[b]	30
Requirement Interaction	30	435
GoalCheck	3	33
ContentionAnalysis	0	32
Other	62	48
Total	99	802

a.DEALSCRIBE added initial requirements and interactions after parsing documents previously produced by the (author) analysts.
b.This snapshot was taken after a new requirement, root, and interactions were added to the "manual" case that was parsed into DEALSCRIBE.

Dialog goal monitoring provides assurance that the dialog protocol is being met. In contrast to operation automation, protocol assurance is directly attributable to dialog goal monitoring. On the other hand, it is more difficult to assess what effect this assurance has on analysts. Nevertheless, the following subjective findings may be of interest until subsequent empirical studies can be conducted.

As analysts, we believe that development goal monitoring provided us with a better understanding of the current development state than we had obtained in our manual case study. Such understanding was gained through warnings that arose in response to goal failures. Once the goal failures were removed, we were assured that the development satisfied the dialog protocol.

As analysts, we believe that DEALSCRIBE provided us with a better visualization of the requirements development. As illustrated in Fig. 3, one can see those statements that fail subgoals of a goal tree. Such visibility of goal failure, with its direct linkage to statements, provides tangible assurance that development goals are being tracked. More generally, the collaborative discussion environment of DEALSCRIBE provides a visualization of development that was not available in our word processor based case studies. For example, the presentation of analysis as a hierarchical discussion (see Fig. 6) provided an understanding of how new analysis fit in with older results.

[12] A statement assertion and its subsequent modification are counted separately. For example, an initial requirement and one modification counts as two requirement statements. Also, some statements may have duplicate contents. For example, two separate conflicts may generate the same resolution, but each would be counted separately.

D. Future Directions

Our research on development goal monitoring is continuing along two directions. First, the functionality of the dialog monitoring system is being extended. Currently, goals are checked and remedied according to a predefined goal tree. While such goal trees can be modified during development, there is no support to do so. A future version of DEALSCRIBE will support dynamic dialog planning. Given predefined remedy operations, with pre- and post-conditions, the planner will check goals and dynamically construct and apply operations to re-establish failed goals. This will simplify the expression and execution of dynamic dialog models. Second, a library of predefined development protocols and associated analyses is being constructed. Future work will focus on expanding this library, proving protocol properties, and integrating the application of various requirements techniques under a common dialog protocol.

VI. CONCLUSIONS

A development goal monitoring system aimed at supporting multi-analyst requirements development has been presented. The conceptual design specifies a multi-user forum of informational and operational statements that can be analyzed and monitored for correspondence to a dialog protocol. The meta-model describing, and used in, the dialog system defines statements whose properties can be formally defined and automatically checked. Such a meta-model facilitates analysis, as well as system extensibility.

An implementation of the development goal monitoring system has been presented. DEALSCRIBE demonstrates support of: a typology of informational and operational statements, checks and remedies of formal goals, operation execution in response to monitored goals, and hypothetical statement assertion, all in a multi-user WWW environment.

An exploratory case study has been presented. It demonstrated that a goal monitoring system can provide automation for a multi-user dialog, assurance about compliance to a dialog protocol, and greater understanding of the requirements development.

This research contributes to requirements management by showing how one can formally specify development goals that can be directly monitored as part of a multi-user dialog. The dialog meta-model provides a concise extensible model for uniformly including informational and operational statements, as well their monitored execution. Development goal monitoring within a collaborative requirements analysis tool can provide a powerful environment for managing development and document inconsistencies.

ACKNOWLEDGMENT

We gratefully acknowledge the help and cooperation of Drs. Colin Potts and Annie Antón for providing documentation of their Inquiry Cycle analysis of the meeting scheduler. We thank Dr. Martin Feather for providing a tutorial on his goal monitoring system, FLEA, that greatly assisted our construction of DEALSCRIBE. We also thank Bala Ramesh, Veda Storey, and the anonymous reviewers for their comments on earlier drafts of this work. Finally, we thank Georgia State University and the College of Business for funding portions of this research.

REFERENCES

[1] Alford, M., Strengthening the Systems Engineering Process, Proceedings of NCOSE, San Jose, CA, 1991.

[2] Antón, A.I., Goal-Based Requirements Analysis, *Second International Conference on Requirements Engineering (ICRE `96)* , *IEEE,* Colorado Springs, Colorado, April 15-18 1996, pp. 136-144.

[3] Barki, H., Hartwick, J., Rethinking the Concept of User Involvement, MIS Quarterly, 13 (1), 1989, pp. 53-61.

[4] Bendifallah, S., and Scacchi, W., Work structures and shifts: An empirical analysis of software specification teamwork. IEEE, *Proceedings of the 11th International Conference on Software Engineering*, 1989, pp. 260-270.
[5] Boehm, B., In, H., Identifying Quality-Requirement Conflicts, IEEE, Software, March, 1996, 25-36.
[6] Brooks, F., *The Mythial Man-Month*, Addison-Wesley, Reading, Mass., 1975.
[7] Ceri S., Gottlob G., Tanca L., Logic programming and databases, Springer-Verlag, 1990.
[8] Checkland, P., Systems Thinking, Systems Practice, John Wiley & Sons, 1981.
[9] Chikofsky, E.J., Rubenstein, B.L., CASE: Reliability Engineering for Information Systems, in *Computer Aided Software Engineering (CASE)*, Ed. Chikofsky,E., IEEE, 1993, pp. 147-153.
[10] Chung, L., Nixon, B.A., Yu, E., Using Non-functional Requirements to Systematically Support Change, In Proceedings of the *Second IEEE International Symposium on Requirements Engineering*, IEEE, March 27-29, York, England, 1995, pp. 132-139.
[11] Chen, M. and Nunamaker, J., The architecture and design of a collaborative environment for systems definition, *Data Base*, Winter/Spring 1991, 22-28.
[12] Curtis, B., Krasner, H., and Iscoe, N., A field study of the software design process for large systems, *Communications of the ACM*, Vol. 31, No. 11 (November 1988), 1268-1287.
[13] Dardenne, A., van Lamsweerde, A., Fickas, S., Goal-Directed Requirements Acquisition, *Science of Computer Programing*, 20 1993, 3-50.
[14] Davy, C. Using CASE to control a large data analysis project. In K. Spurr and P. Layzell (eds.), CASE on Trial. New York: Wiley, 1990, 7-16.
[15] Easterbrook S., Domain modeling with hierarchies of alternative viewpoints, IEEE, International Symposium on Requirement Engineering, January 4-6, 1993, 65-72.
[16] Easterbrook, S., and Nuseibeh, B., *Using ViewPoints for Inconsistency Management*; Software Engineering Journal; IEE/BCS, 1996.
[17] Egyed, A., Boehm, B., USC, *Analysis of Software Requirements Negotiation Behavior Patterns*, USC-CSE-96-504, 1996.
[18] Emmerich, W., Finkelstein, A., Montangero, C., Antonelli, S., Armitage, S. & Stevens, R. “Managing Standards Compliance”, IEEE, Transactions on Software Engineering, *this issue*, 1999.
[19] Faulk, S., Software Requirements: A Tutorial, In. Software Requirements Engineering, Eds Thayer, R.H., Dorfman, H., IEEE, Computer Society Press, Los Alamitos, CA, 1997, pp. 128-149.
[20] Feather, M.S., FLEA : Formal Language for Expressing Assumptions Language Description, June 25, 1997.
[21] Festinger, L., *Conflict, Decision, and Dissonance*, Tavistock Publications, Ltd., London(1964).
[22] Fickas, S., Feather, M.S., Requirements Monitoring in Dynamic Environments, *Proceedings of the 2nd International Symposium on Requirements Engineering*, IEEE Computer Society Press, York, England (March 1995) 140-147.
[23] Gotel, O., Finkelstein, A., Contribution Structures, Proceedings of the Second International Symposium on Requirements Engineering, York, UK, 1995, 100-107.
[24] Graf, D.K., Misic, M.M., The Changing Roles of the Systems Analyst, *Information Resources Management Journal*, 7 (2), Spring 1994, pp. 15-23.
[25] Jacobs, S., Kethers, S., Improving Communication and Decision Making within Quality Function Deployment, *First International Conference on Concurrent Engineering, Research, and Application*, Pittsburgh, USA, August, 1994.
[26] Jarke, M., Gallersdorfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S., ConceptBase - a Deductive Object Manager for Meta Data Management, Journal of Intelligent Information Systems, 4 (2), March 1995, 167-192.
[27] Jarke, M., Eherer, S., Gallersdorfer, R., Jeusfeld, M. A., Staudt, M, ConceptBase - A Deductive Object Base Manager, Informatik V, RWTH Aachen, Ahornstr. 55, D-52056 Aachen, Germany, 93-14, 1993.
[28] Kim, E., Lee, J., An Exploratory Contingency Model of User Participation and MIS Use, Information and Management, 11 (2), 1986, pp. 87-97.
[29] Klein, M., Supporting conflict resolution in cooperative design systems, IEEE, *Transactions on Systems, Man, and Cybernetics*, 21 (6), November 1991, 1379-1390.
[30] Krasner, Herb, Curtis, Bill, and Iscoe, Neil, Communication breakdowns and boundary spanning activities on large programming projects, from *Empirical Studies of Programmers*: Second Workshop (edited by Gary M. Olson, Sylvia Sheppard, and Elliot Soloway), Ablex Publishing Corporation, Norwood, NJ, 47-64. (Conference held in Washing, D.C. on 12/7-8, 1987)
[31] Lee, J., SIBYL: a tool for managing group decision rationale, *Proceedings of the conference on CSCW*, (October 1990)
[32] Lempp, P., Rudolf, L., What Productivity Increases to Expect from a CASE Environment: Results of a User Survey, in *Computer Aided Software Engineering (CASE)*, Ed. Chikofsky, IEEE, 1993, pp. 147-153.

[33] Leventhal, N., Using Groupware to Automate Joint Application Development, Journal of Systems Management, 45 (5), September/October 1995, pp. 16-22.
[34] Liou, Y., Chen, M., Using Group Support Systems and Joint Application Development for Requirements Specification, Journal of Management Information Systems, 10 (3), Winter
[35] Lubars, M., Potts, C., Richter, C., A review of th state of practice in requirements modeling, *First International Symposium on Requirements Engineering*, IEEE, January 4-6 1993.
[36] Lyytinen, K., Hirschheim, R., Information systems failures—a survey and classification of the emperical literature, Oxford Surveys in Information Technology, Vol. 4, Oxford University Press, 1987, pp. 257-309.
[37] Magal, S., Snead, K., The Role of Causal Attributions in Explaining the Link Between User Participation and Information System Success, Information Resources Management Journal, 6 (3), Summer 1993, pp. 19-29.
[38] Maiden, N. Minocha, S. Ryan, M., Hutchings, K., Manning, K., 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, March 19-22, 1997.
[39] Marconi Systems Technology, Requirements and Traceability Managemen, Interactive Development Envrionments, Inc., San Francisco, CA, 1993.
[40] Markus, L., Keil, M., If We Build It, They Will Come: Designing Information Systems That People Want to Use, *Sloan Management Review*, Summer, 1994, pp. 11-25.
[41] Mazza, C., Fairclough, J., Melton, B., De Pablo, D., Scheffer, A., Stevens, R., Software Engineering Standards, Prentice Hall, 1994.
[42] McKeen, J., Guimaraes, T., Successful Strategies for User Participation in Systems Development, JMIS, 14 (2), Fall 1997, pp. 133-150.
[43] Mi, P, Scacchi, W, Process Integration for CASE Environments, IEEE *Software*, Vol. 9(2), 45-53, (March 1992). Reprinted in Computer-Aided Software Engineering (Second Edition), E. Chikovsky (ed.), IEEE Computer Society (1993).
[44] Miller, J., Palaniswami, D., Sheth, A., Kochut, K., Singh, H., WebWork: METEOR's Web-based Workflow Management System, Technical Report #UGA-CS-TR-97-002, Department of Computer Science, University of Georgia, March 1997.
[45] Minker, J., *Logic and Databases: a 20 Year Retrospective*, Invited Keynote Address, Workshop on Logic in Databases, San Miniato, Italy, July 1996
[46] Mullery, G., CORE - a Method for controlled requirements expression, in *Proceedings of the Fourth International Conference on Software Engineering*, IEEE, CS Press, 1979, pp. 126-135.
[47] Mumford, E., Wier, M., *Computer systems in work design—the ETHICS method*, London, Associated Business Press, 1979.
[48] Meyer, B., On formalism in specifications, IEEE, *Software*, 2(1) January 1986, pp. 6-26.
[49] Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M. (1990). Telos: a language for representing knowledge about information systems. ACM Trans. Information Systems 8, 4.
[50] Norman, R.J., Nunamaker, J.F., Jr., CASE Productivity Perceptions of Software Engineering Professionals, *Communications of the ACM*, 32 (9), September 1989, pp. 1102-1108.
[51] Nuseibeh, B., Kramer, J., Finkelstein, A., A Framework for Expressing the Relationship between Multiple Views in Requirements Specification, IEEE, Transactions on Software Engineering, October, 1994, 760-773.
[52] Osterweil, L., Sutton, S., Using Software Technology to Define Workflow Processes, *NSF Workshop on Workflow & Process Automation*, Athens, GA 1996.
[53] Potts, C., Takahashi, K., Anton, A., Inquiry-Based Requirements, Analysis, IEEE, *Software*, pp. 21-32.
[54] QSS Ltd., *Dynamic Object Oriented Requirements System*, Reference Manual, version 2.1, Oxford.
[55] Ramesh, B., Dhar, V., Supporting systems development by capturing deliberations during requirements engineering, IEEE, Transactions on Software Engineering, 1992, pp. 498-510.
[56] Ramesh, B, Representing and Maintaining Process Knowledge for Large-Scale Systems Development, IEEE, *Software*, April 1994, pp. 54-59.
[57] Ramesh, B., Jarke, M., Towards Reference Models for Requirements Traceability, IEEE, Transactions on Softwre Engineering, to appear.
[58] Robbins, S., *Organizational behavior: concepts, controversies, and applications*, Prentice Hall, NJ(1983).
[59] Robey, D., Farrow, D.L., Franz, C.R., Group Process and Conflict in Systems Development, The Institute of Management Sciences, *Management Science*, 35(10), October, 1989, pp. 1172-1191.
[60] Robinson, W.N., Volkov, S., *Conflict-Oriented Requirements Restructuring*, GSU CIS Working Paper 96-15, Georgia State University, Atlanta, GA, September, 1996.
[61] Robinson, W.N., Interactive Decision Support for Requirements Negotiation, *Concurrent Engineering: Research & Applications*, Special Issue on Conflict Management in Concurrent Engineering, The Institute of Concurrent Engineering, 1994 (2) 237-252.

[62] Robinson, W.N., Volkov, S., A Meta-Model for Restructuring Stakeholder Requirements, *Proceedings of the 19th International Conference on Software Engineering*, IEEE Computer Society Press, Boston, USA (May 17-24 1997), pp 140-149.

[63] Robinson, W.N., Volkov, S., Supporting the Negotiation Life-Cycle, ACM, *Communications of the ACM*, to appear.

[64] Robinson, W.N., Pawloski, S., Surfacing Root Requirements Interactions from Inquiry Cycle Requirements Documents, *Third International Conference on Requirements Engineering (ICRE `98)* , *IEEE,* Colorado Springs, Colorado.

[65] Robinson, W.N., Negotiation Behavior During Requirement Specification, *Proceedings of the 12th International Conference on Software Engineering*, IEEE Computer Society Press, Nice, France (March 26-30 1990) 268-276

[66] Sheth A., (Ed.), *Workshop on Workflow & Process Automation*, National Science Founation, Athens, GA 1996.

[67] Spanoudakis, G., Finkelstein, A., Integrating Specifications: A Similarity Reasoning Approach, *Automated Software Engineering Journal*, 2 (4), 1995, pp. 311-342.

[68] Vessey, I., Sravanapudi, A.P., CASE tools as collaborative support technologies, *Communications of the ACM,* 38(1) Jan 1995, pp. 83-95.

[69] van Lamsweerde, A., Letier, E., Integrating Obstacles in Goal-Driven Requirements Engineering, *Proceedings ICSE'98 - 20th International Conference on Software Engineering*, IEEE-ACM (Kyoto, April 98)

[70] van Lamsweerde, A., Darimont, R., Massonet, P., Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt, IEEE, Second International Symposium on Requirements Engineering, March 27-29, 1995, pp. 194-203.

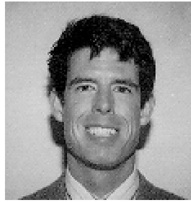
[71] van Lamsweerde, Darimont, R., Massonet, P., The Meeting Scheduler System—Preliminary Definition, Internal Report, University of Louvain, 1993.

[72] Walz, D.B., Elam, J.J., and Curtis, Bill, Inside a software design team: Knowledge acquisition, sharing, and integration, Communications of the ACM, Vol. 36, No. 10 (October 1993), 63-77.

[73] Walz, D.B, Elam, J.J., Krasner, H., A methodology for studying software design teams: An investigation of conflict behaviors in the requirements definition phase, from Empirical Studies of Programmers: Second Workshop (edited by Gary M. Olson, Sylvia Sheppard, and Elliot Soloway), Ablex Publishing Corporation, Norwood, NJ, 83-99. (Conference held in Washington, D.C. on 12/7-8, 1987)

[74] Winograd, T., Flores, F., *Understanding Computers and Cognition*, Addison-Wesley, 1987.

[75] J. Yen and W. Tiao, A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements, in *Proceedings of the Third IEEE International Symposium on Requirements Engineering* (RE'97), January 5-8, 1997.



William N. Robinson received his Ph.D. in computer science from the University of Oregon in March, 1993. He has a M.S. degree in computer science from the University of Oregon and a B.S. degree in computer science from Oregon State University. Dr. Robinson is currently an assistant professor at Georgia State Universities Computer Information Systems Departments. He is Secretary of IFIP Working Group 2.9 (International Federation of Information Processing, Requirements Engineering) and the program chair of the Fourth IEEE Requirements Engineering Symposium (RE'99). Dr. Robinson's research is currently seeking to improve the theory and techniques of requirements engineering—especially, multi-participant requirements analysis. A central part of his research involves the identification and resolution of conflicts.

Suzanne Pawlowski is currently a Ph.D. candidate in Computer Information Systems at Georgia State University. Prior to entering the doctoral program she was a computer scientist at Lawrence Livermore National Laboratory where she was a manager of application development. She received the M.B.A. degree in Management and the B.A. degree in Computer Science from University of California, Berkeley.