Managing Requirements Inconsistency with Development Goal Monitors

William N. Robinson & Suzanne Pawlowski

DEPARTMENT OF
COMPUTER INFORMATION SYSTEMS
GEORGIA STATE UNIVERSITY

September 1, 1997
GSU CIS Working Paper
CIS-97-4
Managing Requirements Inconsistency with Development Goal Monitors

William N. Robinson and Suzanne 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 software requirements during their development can be a complex and difficult task. The requirements can be voluminous, complex, and changing. The analysts, who develop the requirements, can be numerous and changing. Determining the current state of requirements in such a dynamic development environment is challenging. We present techniques to manage requirements in such a dynamic environment.

We define requirements management techniques which uncover, manage, and restructure requirements inconsistencies. Aspects of these techniques have been automated within a multi-user World Wide Web environment, called DEALSCRIBE. The techniques, and their support within DEALSCRIBE, are methodology neutral; they simply assume that requirements development is composed of discrete events and products which can be monitored. Requirements development is then managed by specifying development goals, monitoring for their failure, and applying corrective actions when the goals are not met.

To support requirements inconsistency management, we have defined requirements consistency goals which are monitored using our requirements management techniques. To validate the utility of the approach, we have applied the basic techniques in several case studies. Additionally, we have conducted a new case study using the support of DEALSCRIBE. We draw qualitative conclusions from these case studies that: 1) the general approach of managing requirements development through monitoring development goals helps an analyst gain a clearer understand of the requirements state and helps to focus development; and 2) the techniques specific to uncovering, controlling, and restructuring requirements inconsistencies can reduce an analyst’s effort.

Index Terms—Requirements engineering, inconsistency and conflict management, process modeling and monitoring, meta-modeling, requirements tools.
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 shared understanding, and building consensus. A key challenge facing the analyst is the management and analysis of a dynamic set of requirements as it evolves throughout this process. Although a variety of techniques have been developed to support aspects of this process, support of requirements monitoring has been lacking. In this article, we describe our requirements dialog meta-model, and DEALSCRIBE tool, which were developed to address this need. A key feature of DEALSCRIBE is its ability to monitor the state of requirements development and alert analysts when policy violations arise during the development process.

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 analytic 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[21][55]. 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 research described in this article focuses on a specific, critical issue in tracking and managing the requirements dialog: the monitoring of requirements development goals of consistency. By using a requirements dialog meta-model, as we describe, analysts can benefit from development goal failures alerts which facilitate the development of a requirements document free from conflict.

In this article, we describe our requirements dialog meta-model (§ II). Automated support of the dialog meta-model is presented in section III. Next, instantiations of operations which support the analysis are presented (§ IV). The final sections present our case study summaries (§ V), observations(§ VI), and conclusions(§ VII). However, before we begin, the remainder of this introduction motivates this research and places it in context.

A. Inconsistency: A Driver of Requirements Dialog

Inconsistency is a central driver of the requirements dialog. By understanding and monitoring inconsistency, one can support the management of requirements inconsistency during development. Two basic drivers of requirements inconsistency are: 1) technical and 2) social-political.

Technical drivers of requirements inconsistency generally arise from errors of description; they
include:

- **Voluminous requirements** The sheer size of a requirements document set can lead to inconsistency, such as varied used of terminology. This is especially true as the requirements are modified: one change request can lead to a cascade of other change requests until the requirements reach a more consistent state.

- **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.

- **Changing requirements** As the requirements document is developed, new requirements are added, older ones are updated. As a result, the document is typically in a transitory state where many semantic conflict exists, of which most are expected to be resolved simply by bring them to the current state as (implicitly) understood by the analyst.

*Social-political* drivers of requirements inconsistency arise from differences in goals held by various system stakeholders; they include:

- **Changing and unidentified stakeholders** Analyst report that the initial stakeholder set, defined at project inception, changes as the project progresses. For example, analysts felt that they could understand the system requirements when interacting with actual users; however, such access was often difficult to come by[24]. 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[24]. Such organizational interactions can lead to drastic changes in the requirements.

- **Changing requirements** 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.

- **Changing analysts** Over the life of the project, the composition of team members will change. Consequently, requirements concepts and their expressions will vary over time.

Such drivers are similar to the factors found to be the cause of failures in many information systems[25]. That study concluded that many information system failures could be attributed to poor analysis of important stakeholders. This has been supported in other MIS research[27][42]. Additionally, industry and market-oriented analysts have recognized a need to address multiple, often hidden stakeholders, and their interacting requirements[24].

**B. A Need to Support Analyst 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[5][23][36]; however, they have been less successful in supporting requirements analysis[23]. 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[17]. Thus, analysts will benefit from techniques and tools which 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 [33]. 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 which check for syntactic errors or perform checks of programs inputs and path execu-
tion. However, requirements faults are not been related 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[25][42].

Empirical studies of software development projects have identified a need for issue tracking tools[8][56]. Typical problems include: 1) unresolved issues that do not become obvious until integration testing, and 2) a tendency for specific 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.

C. 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[14]. Feather has produced a working system, called FLEA, that allows one to monitor interesting events defined in a requirements monitoring language[12]. Finkelstein has since illustrated how the technique may be used to monitor process compliance[11]; for example, organizational compliance to ISO 9000 or IEEE process descriptions[28]. Our work on dialog monitoring is derived from these work, but also include 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[7]. 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[40]. Requirements are developed in response to discussions consisting of questions, answers, and assumptions. By tracking these types of dialog elements (and their refinements), dialog is maintained, but inconsistency, ambiguity, and incompleteness are kept in check through specific development operations and requirements analysis (e.g., scenario analysis).

Workflow and process modeling provide some solutions for the management of requirements development[50]. It is possible, for example, to generate a work environment from a hierarchical multi-agent process specification[30]. There has been some attempt to incorporate such process models into CASE tools[29]. However, these tools generally aid process enactment, through constraint enforcement. However, as Leo 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.[38]

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

There are a variety of other projects that 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[6][41][58]; 2) a methodological approach, in which the application of a system development method surfaces conflicts—for example, CORE[31], ETHICS[32], Soft Systems Method[4], ViewPoints[37], and CORA[43]; and 3) a technological approach in which a specific technique, or automation, which can be used to surface requirements conflicts—for example, conflict detection through a collaborative messaging environment[3][18][22], structure-based conflict detection[51], scenario-based conflict surfacing[2][26][40], and conflict classifica-
The dialog meta-model, by virtue of being a meta-model, is neutral to the above approaches. To use the dialog model, a methodology, conflict ontology, and automated techniques can be instantiated as elements of the dialog model. For example, the Inquiry Cycle Model is defined by instantiating the information subtypes of Requirement, Question, Answer, Reason, Decision, and ChangeRequest, as specified in the Inquiry Cycle Model[40]. The dialog meta-model provides the framework by which to instantiate such elements; its implementation in DEALSCRIBE, provides some automation for the definition, execution, monitoring of the dialog.

D. Requirements Development Needs of a Dialog Meta-Model

The design of the requirements dialog meta-model, and its implementation in DEALSCRIBE, were driven by the following requirements development needs:

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

We will show how each of these needs can be supported by the requirements dialog meta-model, and its implementation in DEALSCRIBE.

II. A DIALOG META-MODEL

To support experiments with the automated assistance of dialogs, we have defined a dialog meta-model (DMM) as depicted in figure 1. There are three basic components of the meta-model:

- **Statement Model**
  Statements are added to the dialog by the people, or agents, involved in the dialog. In the dialog

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

---

1 We choose the term dialog meta-model rather than the common term process model, due to our more specific modeling of dialog processes and our use of the meta-model to define other models.
statement model, there are two subtypes of the statement hierarchy:

- **Information.** A passive statement which adds new information to the dialog by reference, or copying, some external information source.
- **Operation.** An active statement which adds new information derived through some computation based on some, or all of, the prior dialog statements.

**Statement History**

The dialog statement history is simply the recorded set of statements which are part of a particular dialog at some point in time. When statements are created, they are said to be **asserted** into the statement history.

**Goal Model**

The dialog goal model is a declarative prescription of the “dialog rules”, indicating such things as the relative order of statements, as well as their content. Examples of dialog goal models include: Roberts Rules of Order, and the software development life-cycle. Enforcement of the dialog goal model may be carried out through statement pre-conditions which restrict the addition of statements to the dialog. Conversely, statements may be unrestricted, but operations can analyze the statement history to determine the degree of compliance to a dialog goal model. In either case, an information or operation statement (subtype) is said to **support** a specific **dialog goal** if: (a) its pre-condition maintains the goal, or (b) its operation (partially) determines the state of the goal.

The dialog meta-model regards a dialog as a stream of statements which fall into passive information and active operations and have some correspondence to the dialog goal model. This kind of meta-model has proven to be quite useful. For example, the meta-model can be refined to define a typical process model, with a distinction of process and product. First, consider each information statement to be a product. Second, consider basic operation statements to be actions within a process model. Third, consider the dialog goal model as the explicit definition of a process model. In fact, the dialog meta-model is a process model with an explicit representation of the process goals and enactment history. As such, we find the dialog meta-model to be suitable for modeling requirements development.

In this article, use of an adaption of the CORA meta-model as it is supported within the dialog meta-model. The aim of the Conflict-Oriented Requirements Analysis (CORA) meta-model is to provide representations and operations useful in analyzing and resolving requirements inconsistencies[43]. The basic entities include: Requirement, Interaction, and Transformation. Using these entities, and their subtypes, one can represent requirements inconsistencies (as interactions) and resolve them through the application of transformations.

To address the management of requirements inconsistencies, we have adapted CORA’s original model to include entities useful in a dialog style of development. These include basic entity refinements, such as the new subtypes of Requirement: InformalRequirement, StructuredRequirement. Additionally, we have added information “mark up” subtypes, including: Note, Question, and Request. These new statement types aid analysts in their dialog about the requirements, and well as their development of the requirements. Finally, we have added new operation statements which provide feedback on the current state of requirements; section IV presents these operations. The application and monitoring of these operations provide a key capability for managing requirements inconsistency.
Fig 2. A portion of a DEALSCRIBE WWW page is shown on the left. Each named radio button indicates a statement type. On the right, a portion of the ConceptBase database is shows the corresponding statement types (as viewed from ConceptBase's graphical browser). Due to space limitations, the OperationType hierarchy was not expanded in the ConceptBase pane.

III. TOOL SUPPORT FOR THE DIALOG META-MODEL

We have developed a tool, called DEALSCRIBE, which supports digital interactions which can be characterized using the dialog meta-model.\(^2\) DEALSCRIBE was created by building upon two existing tools: HyperNews and ConceptBase.

- **HyperNews** provides a discussion system similar to Usenet News, but it has a World Wide Web interface. In each *forum* a user can post *typed* text *messages*. A message may be posted to the forum, or in response to a particular message. A WWW 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 3). HyperNews provides: various views of a forum, user notification of new responses, an email interface, security, and administrative functions.

- **ConceptBase** is a deductive database which provides a concurrent multi-user access to O-Telos objects[19]. All classes, meta classes, instances, attributes, rules, constraints, and queries are

\(^2\) **DEALSCRIBE** is a member in the DEALMAKER suite of tools aimed at assisting collaboration through negotiation in requirements analysis[47] and electronic commerce[45].
uniformly represented as objects. ConceptBase itself operates as a server, while clients, such as ConceptBase’s graphical browser communicate via internet protocols. ConceptBase has shown to be a powerful tool for systems development, partly because of its ability to simultaneously represent and query, instances, classes, and meta-classes[16][34].

In building DEALSCRIBE, we used ConceptBase to define the dialog meta-model and refinements, such as our adaptation of the CORA meta-model. ConceptBase also stores the dialog history as instances of a DMM. The actual interface to the dialog history is managed by an adaptation of HyperNews. It generates statement input and output forms from the definitions of the DMM stored in ConceptBase. Thus, DEALSCRIBE statements can be simple text (as in HyperNews), input forms of typed attributes, or even the result of an operation (e.g., program, or ConceptBase query).

Figure 2 shows screen portions of DEALSCRIBE and ConceptBase’s graphical browser. DEALSCRIBE “Add Message” button types are defined from the corresponding ConceptBase definitions. As statements are added to a particular dialog, they are asserted into the ConceptBase as instances of the statement types shown in the figure.

A. Defining a Dialog Goal Model

To define the “rules of the dialog”, an analyst specifies a set of logical conditions, called dialog goals, about dialog statements. A dialog goal defines desired properties of statements, or their interconnections, possibly over time. For example, consider the goal of having all requirements have a defined user priority. (This could be used to support standard PSS05, which specifies that under incremental development, all requirements will have a user defined priority[28].) The following ConceptBase definition specifies the HasUserPriority goal.

```plaintext
QueryClass HasUserPriority in DialogModel isA Requirement, DialogGoal with
    constraint
    ID_Priority : $ exists x/Priority (this userPriority x) $
end
```

The above definition specifies HasUserPriority as a DialogGoal. The goal is defined as a query about requirements. When ran as a query, it will retrieve all requirements which have a userPriority attribute which is filled is any value of type Priority. (In the ConceptBase query notation, “this” refers to the instance retrieved from the database before the constraint is applied; in this case, a Requirement.)

Complex dialog goals can be created through the constraint language provided by ConceptBase. For example, consider the case where requirements have an associated degree of inconsistency, called contention. We may want to resolve interactions among the most contentious requirements
first. We can specify such a goal as follows:

Class MostContentiousUnresolvedRequirements isA StructuredRequirement with
  constraint
  ConReq : $ not exists gr1/GenerateResolution (gr1 requires this) and
            exists thisCon/Integer (this Contention thisCon) and
            not exists otherReq/StructuredRequirement otherCon/Integer
            ((otherReq Contention otherCon) and
             (otherCon > thisCon) and
             not exists gr2/GenerateResolution (gr2 requires otherReq)) $
end
QueryClass ResolveHighestContentionFirst in DialogGoal isA RequirementInteraction with
  checkModes
    violation : Violation
  constraint
    RHF : $ exists req1,req2/MostContentiousUnresolvedRequirements
          (this requirements r1) and (this requirements r2)) $
end

The above dialog goal definition, 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 make use of the statement history (as captured in ConceptBase) to specify the goal of always selecting unresolved interactions among requirements with the highest contention.

B. Checking the Dialog Goal Model

The dialog goal model can be used to automatically check the statement history for compliance. The dialog goal model consists of a set of goals as specified above. To check compliance, statements need to be compared against the constraints expressed in the goal. Two types of goal modes can be checked: 1) has the goal been achieved, and 2) has the goal been violated. Failures of either type are called, a goal failure. The first is checked by simply running the goal query. The second is checked by finding statements in the statement history which do not meet a goal’s constraints. As shown above in the definition of ResolveHighestContentionFirst, the modes of checking can be specified using checkModes. (Typically, goal violations are of greater concern.) A goal violation query can be automatically constructed by negating a goal’s constraint.³ Such a query can be defined as indicated below:

\[ <\text{class-list}>) = \text{isA}\_classes(G) \]
QueryClass CheckGoalViolation\_G isA <class-list> , GoalViolationCheck with
  constraint
  CheckGoal\_G : $ not (this in G) $
end

Given a goal \( G \), the above shows how one can construct a violation query which is of the same

³ In DEALSCRIBE, violation queries are automatically defined as part of initialization after the dialog model is loaded. However, if a goal constraint is null, then a violation query is not defined because the resulting query would be the same as the goal.
types as specified in goal $G$ (i.e., if $G$ is a Requirement, then CheckGoalViolation$_G$ is a Requirement). However, the constraint indicates that the query should return those instances which do not meet the constraints in the goal $G$.

Violation checking queries, as defined above, can not only determine if a goal is met, but which statements fail the constraint. It is possible to place a goal’s constraints into the constraints of statement definitions. Such integrity constraints would ensure that the dialog goal model is maintained at all times by rejecting statements that do not conform. (DEALSCRIBE allows this.) However, when a statement assertion failed, it would not be possible to determine which of multiple goals the statement violated—when using most database technology. Moreover, no deviations from the dialog goal model would be allowed. So, to enable a more flexible administration of dialog goal models, DEALSCRIBE runs goal checking queries to determine dialog compliance.

C. Defining Statements

Checking the statement history for dialog compliance can itself be considered a dialog operation. In fact, defining the dialog goal or statement models can also be considered dialog operation. Currently, DEALSCRIBE is not used to define the dialog goal or statement model. Instead, models are defined outside of DEALSCRIBE (using a text editor and ConceptBase tools).

Information statements are simply defined as ConceptBase objects. For example, a Structured Requirement with a perspective, mode, and description, could be defined as follows:

```plaintext
Class StructuredRequirement is A Requirement, InformationStatement with
  attribute
    perspective : Perspective;
    mode : Mode;
    description : String
  end
```

From this definition, DEALSCRIBE generates an input form. A user can then fill in, or select, values for attributes of the object. Operation statements are similarly defined. For example, RunAnalysis is (partially) defined as follows:

```plaintext
Class RunAnalysis is A OperationStatement with
  attribute
    query : DialogModelQuery;
    result : Class
  end
```

Like information statements, the object attributes of operation statements may serve as input fields; however, some may serve as output. All operation statements have an associated (Perl) subroutine which is called. After a user fills in the input attributes, statement assertion begins. The subroutine associated with the statement type is executed. It carries out the operation (typically a ConceptBase query) and fills in the output attributes of the object and the statement is asserted. In the above RunAnalysis, the program executes the selected queries and places the returned objects in the result attribute.

D. Defining Monitors

A monitor can be used to continually check dialog compliance against a dialog goal model. In fact, in DEALSCRIBE, any operation statement can be used to monitor the statement history. To do so, 1) a user asserts an operation statement, $S_I$, then 2) a user asserts a StartMonitor statement as a response to $S_I$. The original assertion of $S_I$ allows for the input parameters of $S_I$ to be filled in;
optionally, the operation $S_I$ may execute and assert its results. The assertion of the StartMonitor defines the conditions under which operation $S_I$ will be invoked. DEALSCRIBE will run the operation, according to the monitor parameters, until a StopMonitor is asserted for $S_I$. The statement history, as depicted by DEALSCRIBE in figure 3, indicates: 1) the initial assertion of ModelCheck, 2) the subsequent StartMonitor, 3) the subsequent monitor results, and finally, 4) the StopMonitor statement. Thus, monitoring is divided into two parts: 1) the condition under which the operation will be invoked, and 2) the operation itself. Additionally, the operation may have its own conditions which must be met before results are asserted.

The definition of a monitor specifies under what conditions an operation will be invoked. Commonly, a monitor specifies that an operation shall be invoked after every transaction. In the case of monitoring a goal, this will ensure that a goal violation is immediately detected. However, some operations may be computationally expensive, in either checking applicability conditions or asserting results. In such cases, the monitor can be used to more selectively invoke the operation. Monitors may be run periodically; for example, modulo the statement history count, or chronological time. They may assert new statements every time they are activated, only when the have results, or only when their results are new.

DEALSCRIBE’s use of monitoring can be quite useful. First, basic operations can be automatically run selectively. In addition to simply keeping analysis current, this can include automated synthesis. For example, if a resolution procedure were defined, it could be activated to assert resolution alternatives each time an inconsistency were asserted. Second, goal models can be monitored to alert (or remedy) when compliance lapses. Finally, the goal model itself can include the use of monitoring. For example, it can be specified that requirement contention should be monitored.

---

Fig 3. A portion of a DEALSCRIBE WWW page showing statement headings: <number, icon, title, author date>. The initial CheckModel statement is at the top, followed by a StartMonitor response, and the subsequent monitored responses of three types: CheckModel, RootRequirementsAnalysis, and StopMonitor. The final StopMonitor response ends monitoring of CheckModel. (Responses are shown indented, below, and with the newer statements toward the top.)

Managing Requirements Inconsistency with Development Goal Monitors

GSU Working paper CIS 97-4
periodically, as illustrated below:

```
QueryClass TransactionMonitorContention in DialogGoal isA StartMonitor with
  checkModes
  achievement : Achievement
  constraint
  TMC : $ exists thisOp/RootRequirementAnalysis (this statement thisOp) and
        exists tranPeriod/Integer (this TransactionInterval tranPeriod) and (tranPeriod < 30) and
        not exists stopMon/StopMonitor (stopMon statement thisOp) $
```

The definition of TransactionMonitorContention indicates that it is a StartMonitor operation. The check mode is set to Achievement, indicating that a lack of application of StartMonitor, fulfilling the associated constraints, will result in a monitor invoked operation. The constraints indicate that a StartMonitor statement should be asserted which monitors RootRequirementsAnalysis (an operation), with a transaction period of less than 30—and a StopMonitor for RootRequirementsAnalysis should not have been asserted.

Monitoring of “monitor goals” is accomplished like all monitoring. Consider, for example, monitoring of TransactionMonitorContention. First, an operation statement which analyzes goals achievement and violation, GoalCheck, must be asserted with TransactionMonitorContention as an input. Next, a StartMonitor response to the GoalCheck must be asserted. Whenever the monitor’s condition holds, GoalCheck will be invoked. If TransactionMonitorContention is not achieved, then a GoalCheck statement, indicating the failure, will be asserted. Thus, monitoring itself can be monitored as part of the dialog goal model.

IV. MANAGING INCONSISTENCY

We have developed and applied techniques aimed at assisting the management of requirements inconsistencies. These techniques fall into two approaches based on their basic objective: 1) inconsistency understanding and 2) inconsistency removal. To aid inconsistency understanding, we have developed Root Requirements Analysis[49]. This technique uncovers requirements inconsistencies, analyzes the inconsistencies as a group, and directs analysis to key requirement conflicts. It addresses inconsistency in the sense of requirements incompatibility or requirements conflict. Such conflicts should be resolved prior to construction—even if the resolution is to use an interactive resolver as part of the run-time system. Our second basic approach, called Requirements Restructuring, generates resolutions to requirement conflicts[43][47].

In keeping with the theme of this article, this section shows how Root Requirements Analysis can be incorporated into DEALSCRIBE to assist the management of requirements inconsistency. Root Requirements Analysis is summarized and a DEALSCRIBE dialog goal model for it is defined (§A). Next, we briefly indicates how other related requirements analysis techniques and Requirements Restructuring can also be applied within DEALSCRIBE (§B). These techniques are then illustrated in the following section V, “Case Studies”.

A. Root Requirements Analysis

Two objectives of Root Requirements Analysis are: (1) understanding the relationships among the requirements, and (2) ordering requirements by their degree of conflicting relationships. This information can be used to guide other analyses, such as Requirements Restructuring.

The overall procedure of Root Requirements Analysis is:
1) Identify root requirements that cover all other requirements in the requirement document
2) Identify interactions among root requirements
3) Analyze the root requirement interactions

More specifically: (1) requirements are (manually) generalized to derive root requirements, (2) root requirements are (manually) pairwise compared to derive root requirements interactions, and (3) requirements metrics are automatically 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. The following subsections summarize each step.

Identifying Root Requirements

The objective of root requirement identification is to determine key requirements whose interaction analysis leads to the discovery of significant requirements relationships. While one could exhaustively compare every requirement with every other, in practice, such analysis is not feasible for non-trivial requirements documents. Instead, we seek to identify root requirements which represent key concepts from which other requirements are derived through elaboration. While the binary comparison of such root requirements will not uncover every requirement relationship, it will narrow analysis to key requirements to which further analysis can be applied.

The overall procedure of identifying root requirements is as follows: (1) group requirements into sets by the concepts they reference, (2) order requirements by generality, (3) generate or select the most general requirements for each concept, and (4) repeat steps 1-3 until concept generalizations are not meaningful. The resulting requirements are the root requirements. While it is desirable that the root requirements be a minimal set which cover all other requirements through some set of development relationships, such as elaboration, it is not necessary. In our application, root requirement identification was an informal process aimed at identifying key requirements from which key analyses can be derived.

Identifying Root Requirements Interactions

As Peter G. Neumann notes in his book on Computer Related Risks,

"The satisfaction of a single requirement is difficult enough, but the simultaneous and continued satisfaction of diverse and possibly conflicting requirements is typically much more difficult." — Peter G. Neumann[35].

The objective of identifying root requirements interactions is to surface any such conflicts which can lead to failures during development or operation of the system. For example, individually two requirements may be achieved on a single processor, but simultaneously achieving both can lead to processor thrashing and the achievement of neither. More generally, a requirement may: (1) deplete a shared resource, (2) remove a pre-condition of another requirement, (3) remove the achieved effect of another requirement, or have other interfering actions. We refer to such negative interactions between requirements, as a requirements conflict.

To root identify requirements interactions, each root requirement is exhaustively compared with every other root requirement. For every binary comparison, an analyst subjectively specifies: 1) the relationship type, 2) probability of conflict occurrence during system operation. While such relationships are both subjective and approximate, they have provided a good characterization requirements relationships. In general, such subjective relationships are commonplace among

4 Note that if requirement generalization is not a selective process, then a single requirement (e.g., Thing) would result.
Thus, we apply generalization only when we subjectively deem it conceptually meaningful.
informal requirement techniques[22][39], as well as some formal techniques[6].

Relationship type consists of five qualitative descriptors indicating how two requirements are related to each other; the types are: Very Conflicting, Conflicting, Neutral, Supporting, and Very Supporting. While such requirement interrelationships can be defined formally[1][6] and even automatically derived from formal requirements[9][44], currently, Root Requirements Analysis relies on an subjective determination.

Conflict potential is the subjective assessment of the probability that the requirements conflict will occur in the running system. Consider two requirements, \( R_1 \) and \( R_2 \). If one-third of all system executions result in the achievement of both \( R_1 \) and \( R_2 \) and the other two-thirds results in a system failure, then the probability of conflict occurrence is two-thirds.

Analyzing Root Requirements Interactions

Once the root requirements interactions 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. A completely independent root requirement will have \( n-1 \) Neutral relationships, for \( n \) root requirements. More typically, a root requirements has a mix of conflicting, neutral, and supporting 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.

While other metrics can be derived, we have found relationship count, requirement contention, and average potential conflict to be the most useful. Using these simple computations, the requirements can be rank ordered to guide their efficient resolution. For example, we have found that resolving the most contentious requirement first not only directly resolves one conflict, but often it indirectly resolves others[49]. Thus, resolving high contention requirements first is one of our dialog goals for the Root Requirements Analysis (see ResolveHighestContensionFirst in §II.A).

A Root Requirements Dialog Goal Model

A dialog goal model can be defined for a technique, such as defined in Root Requirements Analysis. The goal model indicates the desired characteristics of processes and products which occur as the statement history is constructed. As such, it can be construed as a methodology prescription for the application of the technique. However, the following Root Requirements Analysis dialog goals are only a part of a methodology. Yet, such partial models are appropriate for the monitoring style of compliance analysis.

The following are four dialog goals for Root Requirements Analysis:

1) DeriveRoots
   Do not have more than 20 requirements which do not have an associated root requirement.

2) DeriveInteractions
   Do not have more than 10 root requirements which do not have an associated interaction.

3) DeriveContention
   Do not have more than 3 new interactions which do not have an associated requirements contention analysis. If so, do contention analysis.

4) ResolveHighestContentionFirst
   Resolve requirements inconsistencies with highest contention first.
Each of the above goals depends on the previous goal in the sequence. The last goal, ResolveHighestContensionFirst, is a follow-up of the first three basic goals of RRA; it was defined in section II.A. The first three goals are actually simpler to defined than ResolveHighestContensionFirst; however, due to the lack of arithmetic in ConceptBase, their definition is slightly baroque.

The first goal simply states the root analysis should take place before too many (20) requirements are defined. The definition would simply involve, count RequirementsWithNoRoot, where RequirementsWithNoRoot indicates those requirements which have not been analyzed. However, using DEALSCRIBE, we must introduce an intermediate ConceptBase “counting goal” which is appropriately interpreted by DEALSCRIBE. Doing so, leads to the following definition:

```
QueryClass RequirementsWithNoRoot isA Requirement with
constrain
  NoRoot : $ not exists r/RootRequirement (r requirements this) $
end
GenericQueryClass CountGoal isA DialogGoal with
parameter
  query : Query;
  count : Integer;
  compare : Integer
end
DialogGoal DeriveRoots isA CountGoal[RequirementsWithNoRoot/query, 20/count, Lesser/compare]
  checkModes
  violation : Violation
end
```

In the above definitions, RequirementsWithNoRoot finds those requirements for which their is no corresponding root requirement. The query, CountGoal, is a special parameterized query whose results are interpreted by DEALSCRIBE. The goal, DeriveRoots, fills in the parameters of CountGoal. The net result is that requirements without associated roots are counted. If the count is less than 20, the goal is achieved; otherwise, it is violated. If monitored, DEALSCRIBE will assert a CheckGoal monitor message (according to the parameters of the monitor) should the goal become violated. The definition of DeriveInteractions is quite similar.

The definition of DeriveContention again takes a similar form, as shown below.

```
QueryClass InteractionWithNoAnalysis isA Interaction with
constrain
  NoAnalysis : $ exists req/Requirement (this requirements req) and not exists con/Integer (req contention con) $ 
end
DialogGoal DeriveContention isA CountGoal[InteractionWithNoAnalysis/query, 3/count, Lesser/compare] |
  checkModes
  violation : Violation
  violationRemedy
  RRA: RootRequirementsAnalysis
end
```

However, there is one additional attribute that is part of the goal. The violationRemedy attribute indicates an operation statement that should be invoked automatically if a violation is observed as part of monitoring. Upon violation, a remedy operation is passed the dialog goal and results of the violation checking query. In the case of the RootRequirementsAnalysis, it simply ignores the input, updates the contention attribute of all requirements, and as executes asserts its RootRequirementsAnalysis statement (see figure 3).

Finally, a Root Requirements model can be defined to consist of the above four goals as illus-
trated below.

DialogModel RRA_GoalModel with
  goals
     TMC : TransactionMonitorContention;
     DR : DriveRoots;
     DC : DriveContention;
     RHCF : ResolveHighestContentionFirst
end

Once so defined, this goal model may be selected as part of the input to run the ModelCheck operation. Thus, multiple goal models can co-exist and can be monitored at different times.

B. Other Requirements Analyses and Operations

As illustrated through this section, the dialog meta-model, as supported in DEALSCRIBE, provides a convenient means of experimenting with monitoring of requirements development goals. Goals are expressed as logical formula over the assertion of the information and operation statements into the statement history. Once the O-Telos logic is understood, it is relatively simple to define new statements, goals, and monitors. For example, to incorporate aspects of the PSS05 standard in a working goal model, the following was done[28]:

1) Add userPriority as an attribute to the Requirement information statement type.
2) Define LimitEmptyPriorities as a dialog goal which seeks limit the percentage of requirements without a user priority to less than 20 percent. Its definition is similar to that of DeriveRoots, but it uses a goal type that computes percentages.

Based on such small steps, we have found it relatively simple to experiment with different goal models.

The monitoring facility of DEALSCRIBE provides a means to incorporate active monitors. Such monitors do more than signal that a goal has been violated. As illustrated in the DeriveContention goal, active monitors can initiate operations. For example, a DeriveResolution goal can be defined which, upon violation of a goal of consistency, invokes a resolution generation operation which asserts alternative resolutions for an inconsistency. Such monitors, judiciously asserting suggestions in the background, may provide a means to automated development where analyst are opposed to more direct assistance.

V. CASE STUDIES

Case studies have been conducted to assess the utility of the DEALSCRIBE implementation of the dialog meta-model. Specifically, Root Requirements Analysis (§IV.A) was applied to one case both without (§B) and with (§C) the use of DEALSCRIBE. Thus, the case studies help assess the utility of DEALSCRIBE, as well as the dependence of Root Requirements Analysis on a particular tool set. But, before discussing the case studies, the distributed meeting scheduler case is summarized.

A. Requirements for a Distributed Meeting Scheduler

To assess requirements analysis techniques and their tool support, we have repeatedly analyzed the case of the distributed meeting scheduler requirements. The meeting scheduler case is useful because of: (1) the complex requirements interactions which, depending on how they are addressed, lead to considerable variation in the resulting implementations; (2) the availability of a
widely circulated compact, yet rich, requirements document[53]; and (3) the publication of prior analysis of the case[40][52]—including our own[43][44][47]. Hence, this case allows us, and others, to compare analyses[13].

The general problem of the meeting scheduler can be summarized by the introduction to the requirements[53]

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

The remaining requirements of the four page description refine the roles of the meeting scheduler and participants.

B. Root Requirements Analysis of Inquiry Cycle Documents

To assess the utility of Root Requirements Analysis, we applied the method to an established requirements engineering case, that of the distributed meeting scheduler. The objective of the case-study was to assess two hypotheses: (1) could Root Requirements Analysis be easily incorporated into an existing methodology?, and (2) could Root Requirements Analysis add value by uncovering requirement relationships? Fortunately, we obtained access to analysis documents generated during an the Potts et. al. application of the Inquiry Cycle to the distributed meeting scheduler problem[40]. By applying Root Requirements Analysis to the Inquiry Cycle discussion documents, we were able to assess both hypotheses.

Given the Inquiry Cycle analysis, we considered two ways to apply Root Requirements Analysis. First, the original requirements could analyzed; such a case-study would result in a direct comparison between the Inquiry Cycle and Root Requirements Analysis. Second, the requirements discussion of the Inquiry Cycle could be analyzed. For the initial study, we choose the second approach, as it provided an illustration of how Root Requirements Analysis could augment another method[49]. However, the subsequent case study included both analyses within DEALSCRIBE.

Figure 4 illustrates the result of applying the Inquiry Cycle model to the distributed the meeting scheduler. The case produced 33 questions of the original requirements; 40 answers to those questions; 38 changes to the requirements; and 18 reasons for the changes that were made.

Root Requirements Analysis was conducted using processor and a spreadsheet program. As illustrated in figure 4, it led to the discovery of 23 very conflicting and 49 conflicting relationships. The basic relationships for each root requirement are shown as a percentage of all relationships in figure 5.

Root Requirements Analysis was useful in managing requirements interactions. As described in section IV.A, the relationship count, requirement contention, and average potential conflict can be used to determine which requirement conflict should be resolved first. In particular, we have found it beneficial to resolve the most contentious requirement first. Thus, figure 5 shows that $R_8$ and $R_3$ are among the most contentious of all root requirements. However, of those that directly interact with each other (see ResolveHighestContensionFirst in §III.A) $R_8$ and $R_{13}$ have the highest contention. Thus, their interaction was considered first as part of conflict resolution.

C. Assisted Root Requirements Analysis

To assess the utility of DEALSCRIBE, we applied the same Root Requirements Analysis to the
Fig 4. Results of applying Root Requirements Analysis to the Inquiry Cycle discussion. In the Inquiry Cycle, ovals indicate the number of unique instances of a type, while arcs indicate the flow of analysis within the Inquiry Cycle. The Root Requirements and graph of relationship counts by type was created from Root Requirements Analysis.

The objective of this second case-study was to determine: 1) if DEALSCRIBE could automate the basic metric analyses of Root Requirements Analysis, and 2) if dialog goal monitoring would be useful. The first objective was readily affirmed—DEALSCRIBE can automate Root Requirements Analysis. The second objective is more subjective and will require empirical studies. However, based on our use of DEALSCRIBE, we found goal monitoring to be of considerable assistance in managing requirements development.

The automation study duplicated the manual study, but with the use of DEALSCRIBE. First, monitoring of the Root Requirements Analysis dialog goal model was started (§IV.A). Second, the

Fig 5. Graph of root requirements interactions. The percentage that each requirement participates with all other requirements for five relationship types (Very Conflicting, Conflicting, Neutral, Supporting, and Very Supporting) are presented in an additive "stacked" graph; ordered by increasingly negative interactions.
Fig 6. A comparison of applying Root Requirements Analysis to: the a) Inquiry Cycle (IC) discussion, and the b) original requirements, showing the relative number of root requirements and requirement conflicts.

previously identified text requirements of the inquiry cycle discussion, root requirements, and interactions were automatically parsed and asserted into DEALSCRIBE. As statement were asserted into DEALSCRIBE, goal failures were recognized and new information and remedies were automatically asserted (see figure 3).

The Root Requirements Analysis results were the same as figure 4—as expected. However, the analyses were automatically asserted in response to goal failures which incrementally occurred during input. Thus, DEALSCRIBE maintained the Root Requirements Analysis metrics, including the causes of goal failure.

Another case study using DEALSCRIBE was conducted. Rather than analyze the requirements derived from the Inquiry Cycle discussion, as was done previously, the original meeting scheduler requirements were analyzed. As is turns out, both documents have some common root requirements. Consequently, DEALSCRIBE could quickly derive the rather surprising results illustrated in figure 6.

What is surprising is that analyzing the original 53 requirements uncovered nearly as many conflicts as analyzing the Inquiry Cycle discussion. A priori, we hypothesized that, as a record of stakeholder interaction, the Inquiry Cycle discussion would be richer in information—especially conflicting requirements. As it turned out, the root requirements were nearly evenly distributed across three sets: 10 from the Inquiry Cycle discussion, 11 from the original requirements, and 20 in both. Similarly, the conflicting interactions were nearly evenly distributed across three sets: 16 from the Inquiry Cycle discussion, 11 from the original requirements, and 55 in both; additionally, there were 8 derived from interactions between the original and Inquiry Cycle roots.

VI. OBSERVATIONS

From the case studies, we observe that Root Requirements Analysis in useful, independent of tool support. However, automated monitoring of the Root Requirements Analysis development goals can significantly clarify the development status and reduce the effort analysts.

A. Root Requirements Analysis

Root Requirements Analysis has been valuable in managing the development of requirements. The technique can be applied to requirements irrespective of their form or refinement. It provides:
• an ordering of the most conflicting and interacting requirements
• requirements dependencies across the whole system
• summary information that is easily understandable through tables and graphs.

The information that Root Requirements Analysis provides insight as to where the requirements development effort should be applied. For example, if one seeks to: 1) reduce the overall number
of requirements conflicts (i.e., seek *monotonically decreasing contention*), and 2) reduce the number of prior resolutions that must be reconsidered (i.e., seek *minimize resolution backtracking*), then one should resolve the most contentious interacting requirements first. Root Requirements Analysis can find such requirements by ordering conflict interactions by the degree of requirements contention.

Root Requirements Analysis can provide a high level understanding of the requirements interactions. Through the root identification process of generalization and the subsequent interaction identification, higher-level interaction patterns and issues emerged. For example, in the meeting scheduler, it became apparent that many root requirements had interactions concerning: 1) accurate meeting planning data and 2) the need to complete the meeting scheduling process in a timely way. Issues at this level of abstraction can be brought to the stakeholders for discussion and negotiation, providing the analysts with guidance about relative priorities that can be used in the conflict resolution process. Working on issues at this higher level requires significantly less time than reviewing each individual conflict, and promotes consistent decision-making throughout the conflict resolution process.

### B. DEALSCRIBE's Dialog Modeling

DEALSCRIBE can be valuable in managing the development of requirements. DEALSCRIBE provides:

- modeling of informational and operational statements
- modeling of dialog goals
- monitoring of goal failures
- monitored analyses
- concurrent multi-user WWW interface to the dialog

DEALSCRIBE can manage dialogs where asserted statements can be represented as hierarchies of informational and operational statements.

Applying DEALSCRIBE to the meeting scheduler case helped the analysts gain a clearer understanding of the requirements state and helped to focus development. Once the Root Requirements dialog model was defined, the root analyses (metrics) were automatically derived by DEALSCRIBE. Thus, rederiving the original analysis in DEALSCRIBE took essentially no effort. In the original manual Root Requirements Analysis, analysts had to coordinate their work and refer to a common spreadsheet to prevent a duplication of effort. In contrast, DEALSCRIBE's dialog view and dialog model monitoring provided a development overview which facilitated multi-user coordination. Finally, the meta-modeling supported by DEALSCRIBE facilitated a continual refinement of the Root Requirements dialog model; for example, after an update of the model, DEALSCRIBE derived a new WWW dialog interface.

### VII. CONCLUSIONS

Applying the dialog meta-model, as implemented in DEALSCRIBE, has demonstrated the utility of actively monitoring development goals—specifically, goals of reducing requirements inconsistency. The success of the dialog meta-model can be partly attributed to the simple WWW interface and various dialog views provided by DEALSCRIBE. However, the goal-based monitoring of the dialog is the key feature. By activating dialog monitors, analysts can be assured that they will be alerted if their process or product goals fail. Moreover, if specified as a goal, remedies can be automatically applied to goal failures. Such active assessment of development goals helps to overcome
the chaos that emerges from the dynamic environment of multi-stakeholder analysis and voluminous, complex, and changing requirements.

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 also thank Dr. Martin Feather for providing a tutorial on his goal monitoring system, FLEA, which greatly assisted our construction of DEALSCRIBE. Finally, we thank Georgia State University and the College of Business for funding portions of this research.

REFERENCES

Olson, Sylvia Sheppard, and Elliot Soloway). Ables Publishing Corporation, Norwood, NJ, 47-64. (Conference held in Washington, D.C. on 12/7-8, 1987)


[57] Yakemovic, K., Conklin, J., Experience with the gIBIS model in a corporate setting, Proceedings of CSCW 90, Los Angeles, 1990.