University of Wollongong
Research Online

Faculty of Informatics - Papers (Archive)

Faculty of Engineering and Information Sciences

November 2000

Formal tools for managing inconsistency and change in RE

Aditya K. Ghose
University of Wollongong, aditya@uow.edu.au

Follow this and additional works at: https://ro.uow.edu.au/infopapers

 Part of the Physical Sciences and Mathematics Commons

Recommended Citation

Ghose, Aditya K.: Formal tools for managing inconsistency and change in RE 2000.
https://ro.uow.edu.au/infopapers/211

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

Formal tools for managing inconsistency and change in RE

Abstract

Dealing with inconsistencies and change in requirements engineering (RE) is known to be a difficult problem. We propose a formal, integrated approach to inconsistency handling and requirements evolution with a focus on providing automated support. We define a novel representation scheme that is expressive and able to maintain several key semantic distinctions. Based on this scheme, we define a toolkit of inconsistency handling technique. We define a principled process for evolving such specifications, with minimal computational cost and user intervention. Finally, we describe the REFORM system which implements some of these techniques.

Disciplines

Physical Sciences and Mathematics

Publication Details

This paper originally appeared as: Ghose, AK, Formal tools for managing inconsistency and change in RE, Tenth International Workshop on Software Specification and Design, 5-7 November 2000, 171-181.

Copyright IEEE 2000.

This conference paper is available at Research Online: https://ro.uow.edu.au/infopapers/211

Formal tools for managing inconsistency and change in RE

Aditya K. Ghose
Decision Systems Lab
Department of Information Systems
University of Wollongong
NSW 2522 Australia
aditya@uow.edu.au

ABSTRACT
Dealing with inconsistencies and change in requirements engineering is known to be a difficult problem. We propose a formal, integrated approach to inconsistency handling and requirements evolution with a focus on providing automated support. We define a novel representation scheme that is expressive and able to maintain several key semantic distinctions. Based on this scheme, we define a toolkit of inconsistency handling technique. We define a principled process for evolving such specifications, with minimal computational cost and user intervention. Finally, we describe the REFORM system which implements some of these techniques.

Keywords
Requirements engineering, formal methods.

1 INTRODUCTION

Understanding and supporting the process of requirements evolution is an important and difficult problem. Fundamentally, requirements evolution involves updating a description of user requirements for a target system to accommodate new requirements or to remove existing ones. Such changes may become necessary because of changes in the real-world context in which the proposed system would be situated or because of changes in stakeholder perceptions of the proposed system. A fundamental problem in supporting requirements evolution is *inconsistency handling*, i.e., dealing with situations where new requirements contradict existing ones (see [16] for a discussion of current issues). Managing inconsistent requirements (i.e., situations where the given set of requirements cannot be simultaneously satisfied) is an important problem in its own right, and any solution to the requirements evolution problem must be built on a framework for inconsistency handling. The problems of requirements evolution and inconsistency handling are closely related to several other questions that any comprehensive framework for requirements engineering must address. The first of these relates to *supporting multiple sets of stakeholders* in multi-perspective software development, who may have distinct, and often contradictory *viewpoints* on the requirements of the proposed system. The second issue involves providing support for *non-functional requirements* or *software quality factors* which closely interact with, and often contradict, functional requirements. The question of supporting *requirements rationale* is closely related. The third question relates to *requirements reuse*, based on the notion that instead of discarding requirements in the process of evolution or inconsistency handling (as several existing frameworks tend to do), it would be better to retain them in anticipation of future reuse, given that requirements are expensive to acquire/elicit.

Part 1 of the example in the appendix illustrates several distinct ways in which conflicts may arise. Functional requirements may contradict each other and may contradict non-functional requirements (the divergent pulls of performance and functionality goals is a common feature of systems development). Goals that are otherwise consistent may be in conflict because of conflicting rationale. We argue that the following features are essential in any framework for handling inconsistent requirements. *First*, it must support the distinction between *essential* and *tentative* requirements in specifications (and more generally, a partitioning of a specification based on levels of priority). In general, one would be more willing to discard a requirement labelled as tentative as opposed to one that is labelled as essential if forced to discard requirements to maintain consistency in a specification. *Second*, it must support a representation scheme that makes explicit the connection between a requirement and the conditions/assumptions/justifications that the satisfaction of the requirement is contingent on. *Third*, it must support a domain-independent facility for making explicit the trade-offs involved when requirements must be discarded to make a specification consistent. In other words, it should be possible to generate every *maximal consistent subset* of an inconsistent specification.

We also argue that any framework for supporting re-

0-7695-0884-7/00 $10.00 © 2000 IEEE

quirements evolution, as well as tools based on such a framework, must satisfy the following criteria. *First*, it must ensure that every evolution step makes minimal change to a specification, along the lines of a similar condition imposed on AI theory change operators [1]. *Second*, when a change step makes it necessary to discard some requirements, it must be based on a detailed trade-off analysis that weighs the cost of discarding the requirements against the cost of ignoring the change request. This ensures that more important requirements are not discarded to accommodate a relatively less important requirement (such as one specified by a low-priority stakeholder). *Third*, it must support a *deferred commitment strategy*, i.e., one which ensures that any choice amongst multiple candidate outcomes is delayed as far as possible to ensure that no premature commitments are made that may turn out to be poor choices in retrospect (choices amongst candidate outcomes may become necessary when an evolution step renders a specification potentially inconsistent and multiple maximal consistent subsets of the specification exist). *Fourth*, it must support requirements reuse, given that requirements are expensive to acquire/elicit. This entails that requirements that would otherwise be discarded in an evolution step are maintained in a background store in anticipation of future reuse.

This paper presents a formal framework with these characteristics. The representation scheme permits us to explicitly represent the interaction between functional and extra-functional goals and their rationale. We provide approaches to consistency handling which generalize earlier approaches to consistency handling, with a focus on providing automated support. We suggest an inexact but practical approach to incorporating elements of system behaviour in the inconsistency handling exercise by recording *critical states* and *trajectories*. Our goal is to explore the extent to which automated support may be provided for managing inconsistency and change, while deploying hard-coded or user-determined criteria for inconsistency resolution in a principled manner (possibly through direct user interaction in the resolution process). To this end, we describe the REFORM system and outline heuristics used to overcome the significant computational bottlenecks inherent in such a problem. The essence of our proposal is independent of a specific choice of an underlying requirements specification language. The only requirement is that the semantics of the language provide a clear notion of consistency.

There is a large body of earlier work that this research builds on. Balzer [2] introduced the notion of *pollution markers* as an approach to tolerating and managing inconsistency in specifications. Tsai proposed the use of non-monotonic logics in resolving inconsistencies in specifications [19] while similar ideas were also explored by Ryan [18]. The *Viewpoints* framework [6] [14] [5] [4] supports multi-perspective development (with multiple sets of stakeholders) by allowing explicit "viewpoints" which hold partial specifications, described and developed using different representation schemes and development strategies. Individual viewpoints are required to be internally consistent while inconsistencies arising between pairs of distinct viewpoints (the authors suggest translation into a uniform logical language for detecting inconsistencies) are removed by invoking meta-level inconsistency handling rules. Mylopoulos, Chung and Nixon [13] present a framework for representing and decomposing non-functional requirements, as well as a process model for using non-functional requirements to guide and justify design decisions. Our focus in this paper is distinct from theirs, in that we seek to manage requirements evolution and the inconsistencies arising from the interaction between functional and non-functional requirements, but it is easy to see their framework fitting in immediately downstream from ours in the software life-cycle. Lamsweerde *et al* [20] explore a wide range of categories of inconsistency in the context of the KAOS framework - this paper takes these results as the starting point. Wiels and Easterbrook [21] have defined evolution techniques based on category theory, while Nuseibeh and Russo [15] have used abductive logic programming. Heitmeyer et al [11] have defined inconsistency handling techniques in the context of tabular notations. Hunter and Nuseibeh [12] present a framework for representing specifications using a logic with a paraconsistent flavour, with some common intuitions with our current work. This work extends our earlier work on using frameworks inspired by non-monotonic logics and logics of theory change to support inconsistency and evolution management [22] [7] [8], by providing a richer representation framework and a broader repertoire of inconsistency handling and evolution operators and by describing an implemented tool. A major case study has largely validated the results discussed here, but is ommitted here for space constraints.

2 REPRESENTATION ISSUES

In the following, we assume a formal first-order language (possibly augmented with temporal operators) for representing both the domain theory and system goals (but note that much of the following discussion applies to any language that comes with a well-defined notion of consistency). A good example of such a formal language is the language used in the formal assertion layer in KAOS [3] [20].

We define a *requirements specification* to be a 5-tuple $< D, E_{FR}, T_{FR}, E_{NFR}, T_{NFR} >$ where:

- D is the *domain theory* and consists of the following two components:
  - A set of *domain invariants* D_{inv}

– A set of *domain trajectories* D_{traj}

- A set E_{FR} of *essential functional requirements.*
- A set T_{FR} of *tentative functional requirements.*
- A set E_{NFR} of *essential non-functional requirements.*
- A set T_{NFR} of *tentative non-functional requirements.*
- $E_{FR} \cup E_{NFR} \cup D_{inv} \cup t_i$ is consistent for each trajectory t_i contained in D_{traj}. In the context of the discussion and examples in this paper, we shall require a weaker condition: $E_{FR} \cup E_{NFR} \cup D_{inv} \cup s_i$ is consistent for each state s_i contained in D_{traj}, as explained below.

D_{inv} is assumed to be any theory expressed in the underlying formal language. The set of domain trajectories D_{traj} would ideally be an oracle capable of generating all system *behaviours* or *scenarios* (in the sense of [3] [20]), i.e., the (non-deterministic) set of possible sequences of system states. In practice, we are only interested in abstractions of system behaviour - it may be sufficient to verify consistency of requirements against a limited set of *critical states.* One way of achieving this is to explicitly store a set of *critical trajectories*, each of which is a sequence of critical states. The cognitive and computational demands for doing this are no greater than those for generating, for instance, UML state diagrams. We will therefore assume that D_{traj} is a set of (finite) sequences of states (i.e., trajectories) of the form $[s_1, \ldots, s_n]$ where each state s_i is a set of assertions in the underlying formal language providing a (possibly partial) description of a system critical state. We avoid temporal operators in our example for simplicity, hence consistency with trajectories reduces to consistency with individual states.

E_{FR}, T_{FR}, E_{NFR} and T_{NFR} are sets of *justified requirements*, where a justified requirement is a pair written as $\alpha : \beta$, with α and β both representing sentences in the underlying formal language. α represents a requirement (i.e., a goal), while β represents its justification. When α represents a functional (resp. non-functional) requirement, $\alpha : \beta$ represents a functional (resp. non-functional) justified requirement. For a functional requirement α, a justification consists of what is otherwise referred to as a *requirements rationale* and might include, for instance, the functional and performance goals that the requirement is intended to support. For a non-functional requirement α, the justification similarly represents its rationale, stated in terms of the functional and performance goals it is intended to support. Essential justified requirements are treated as inviolate at any given point in time, although they may be brought into question over time (as a consequence of requirements evolution). Tentative justified requirements may be violated; our intent is to satisfy as many of them as is consistently possible at any given point in time. We permit α to be any well-formed formula in the underlying language (i.e., α is not restricted to consist of atomic goal assertions, but can also include definitions of these goals). β is similarly any well-formed formula in the underlying language. The final condition requires that at any given point in time, all of the inviolate goals together with the domain invariants are consistent in every critical system trajectory (state, for the purposes of this paper) contained in D_{traj}. The partitioning of the sets of essential and tentative requirements on the basis of whether they refer to functional or non-functional requirements is not essential from a computational perspective, but potentially useful. The partitioning supports an important cognitive and semantic distinction. In addition, it is possible to define variants of the inconsistency handling operators described below which are biased towards satisfying functional requirements over non-functional requirements or vice versa. The reader is referred to Part 2 of the example in the appendix for instances of a requirements specification and applications of the inconsistency handling techniques defined below.

We first need to understand the semantics of consistency of a set of requirements/goals relative to a requirements specification. We shall define the notion of *r-consistency* of a set of justified requirements $R = \{\alpha_1 : \beta_1, \ldots, \alpha_n : \beta_n\}$ with respect to a requirements specification $S =< (D_{inv}, D_{traj}), E_{FR}, T_{FR}, E_{NFR}, T_{NFR} >$ as follows: R is *r-consistent* with respect to S if and only if $\alpha 1 \cup \ldots \cup \alpha n \cup \beta_1 \cup \ldots \cup \beta_n \cup D_{inv} \cup t_i$ is satisfiable for every trajectory $t_i \in D_{traj}$. Our examples in this paper do not involve temporal operators, hence we check for consistency with individual states in D_{traj} instead of entire trajectories. Thus, a set of requirements is r-consistent relative to a given specification when the requirements, together with their justifications are consistent with the domain invariants together with each critical state description (taken individually). Inconsistencies can be detected and resolved in two complementary modes. First, we may identify (as in [20]) minimal sources of inconsistency (or *min-conflict sets*), thus focusing attention on requirements that must be modified or discarded. Formally, a min-conflict set I of a given requirements specification $S =< (D_{inv}, D_{traj}), E_{FR}, T_{FR}, E_{NFR}, T_{NFR} >$ is any set satisfying the following properties:

- $I \subseteq T_{FR} \cup T_{NFR}$
- I is r-inconsistent relative to S
- Any $I' \subset I$ is r-consistent relative to S

In general, multiple min-conflict sets may exist for a

given requirements specification. A maximal consistent subset of goals can be obtained by removing a smallest subset of $T_{FR} \cup T_{NFR}$ that intersects with each minimal inconsistent subset of the requirements specification. In general, multiple such smallest subsets which intersect with each min-conflict set might exist, leading to multiple possible maximal consistent subsets of goals (more on this below).

Second, we may identify (as in [12] and [8, 7]) maximal consistent subsets (we shall refer to these as *r-maximal sets*) of the total set of specified requirements. It may not be necessary to generate each of these sets - the first r-maximal set that all stakeholder groups agree on represents a successful resolution of inconsistencies in a specification. Formally, an r-maximal set M for a specification $S =< (D_{inv}, D_{traj}), E_{FR}, T_{FR}, E_{NFR}, T_{NFR} >$ is any set satisfying the following properties:

- $M \subseteq E_{FR} \cup T_{FR} \cup E_{NFR} \cup T_{NFR}$
- $(E_{FR} \cup E_{NFR}) \subseteq M$
- M is r-consistent relative to S.
- For every M' such that $M \subset M' \subseteq E_{FR} \cup E_{NFR} \cup T_{FR} \cup T_{NFR}$, M' is r-inconsistent relative to S.

As noted above, these two approaches to inconsistency handling are complementary and may be defined in terms of each other (users are thus free to select their preferred approach). Formally, given a requirements specification S as defined above, a *minimal hit-set* H is any set such that:

- $H \subseteq T_{FR} \cup T_{NFR}$
- $H \cap I \neq \emptyset$ for each minimally inconsistent subset I of S.
- There exists no $H' \subset H$ that satisfies the above two conditions.

Then we can show that for every minimal hit-set H, there exists a maximal consistent subset M of S such that $M = E_{FR} \cup E_{NFR} \cup ((T_{FR} \cup T_{NFR}) - H)$. Conversely, for every r-maximal set M of S, there exists a minimal hit-set H satisfying the condition above.

In our formulation of the inconsistency handling process above, an outcome is a r-maximal set of a given specification, selected from amongst the possibly many r-maximal sets that may exist via the application of a choice function. We are thus interested in the class of *outcome choice functions* which take as input a set of r-maximal sets of a specification and produce as output an element of this set. Formally, an *outcome choice function* c_o is defined as:

$$c_o : 2^{\mathcal{M}} \rightarrow \mathcal{M}$$

where $\mathcal{M}$ is the class of r-maximal sets. Beyond requiring this formal structure, we leave the outcome choice function undefined - such choice functions are likely to be domain-dependent, although useful domain-independent strategies for designing such functions can be formulated (such as those that incorporate priorities amongst individual goals, or organizational precedences amongst stakeholders). We believe that the outcome choice function should be dynamic and context-sensitive, reflecting selection criteria that are relevant at the time that it is applied.

Alternative formulations of the outcome of the inconsistency handling process are also worth considering. Instead of seeking maximal (with respect to set inclusion) subsets that are consistent, one could seek consistent subsets of maximal cardinality (this is relevant in many commercial settings where the number of satisfied requirements is critical and plays an important role in the project costing exercise). In settings where goals come with measures of utility, consistent subsets that maximize utility may be of interest.

There are three features in our representation and inconsistency handling scheme that are particularly useful. First, we use a collection of complete critical state descriptions in D_{traj} as a relatively efficient abstraction of an oracle that generates all system behaviours. Second, we support a distinction between essential and tentative goals such that the outcome of an inconsistency handling exercise (an r-maximal set of a specification) satisfies all essential goals and as many tentative goals as is consistently possible. Finally, it is easy to enforce integrity constraints. An integrity constraint $\neg a$ is enforced simply by adding a justified requirement $\emptyset : \neg a$ to either E_{FR} or E_{NFR}. This guarantees that no r-maximal set of the resulting specification will entail the assertion a. There are other useful properties such as the guaranteed existence of r-maximal sets and the orthogonality of such subsets (i.e., distinct r-maximal sets exist if and only if they are r-inconsistent). We omit these details here for brevity and point the reader to [9] for formal statements and proofs.

The discussion above assumes that every assertion representing a requirement is atomic. Modifications, if any, are assumed to be manually performed ([20] provides a detailed set of strategies for this purpose). In actual fact, automatic support can be provided for certain classes of modifications, specifically for those involving *goal weakening* [20]. The underlying intuition is that in some situations, an assertion that is inconsistent with another set of assertions may not have to discarded entirely, but may be modified so as to maximize the *extension* of the original assertion that can be consistently retained. One obvious case where automated support can be provided involves maximizing the extension of universally quantified assertions over the domain. The

intuitions underlying such modifications are the same as those in several approaches to default reasoning (such as hypothetical reasoning, as exemplified by the THEORIST system [17]). When a universally quantified assertion $\forall x P(x)$ contradicts the assertion $\neg P(a)$, it is possible to avoid completely discarding either of these two assertions, by maximizing those portions of the extensions of the original assertions that can be consistently retained by explicitly introducing an exception to the universally quantified assertion. The resulting assertion, $\forall x\ x \neq a \rightarrow P(x)$ is consistent with $\neg P(a)$. Providing computational support for this kind of modification is expensive, requiring an exponential number of (exponential time) satisfiability checks. Automated support can also be provided for certain special cases of the *temporal relaxation* strategies discussed in [20].

Our approach to inconsistency handling generalizes several of the distinct classes of inconsistency identified in [20], except process-level deviations, terminology clashes, designation clashes and structure clashes. An *instance-level deviation* is detected if the state transition leading to the deviation results in a critical state contained in D_{traj}. Under the condition that the boundary condition involved in a *divergence* occurs in a critical state represented in D_{traj}, a minimal set of assertions over which a divergence occurs corresponds directly to a min-conflict set of the specification. *Obstructions*, which are special cases of divergences, are thus also subsumed by the notion of min-conflict sets, under the same conditions. The set of assertions over which a *conflict* occurs also corresponds directly to a min-conflict set. As a special case of divergence, *competition* is also subsumed by our approach. In addition, the method for relaxing universally quantified goals described above can be used to resolve this category of inconsistency.

Requirements engineering is an inherently social phenomenon, and it is useful to view the RE process as rational social decision-making. This involves treating each distinct perspective as a distinct agent, with each agent seeking to maximize its own utility. One way of formulating an agent's utility function in the context of an inconsistency handling exercise is to assign higher utilities to outcomes that satisfy larger subsets of an agent's set of goals. A socially rational outcome must then be pareto-optimal. Viewing each stakeholder group as a distinct agent, it is possible to show that an r-maximal set of a requirements specification represents a pareto-optimal outcome of the social decision process. We have argued above that a stakeholder group's functional and non-functional requirements often represent conflicting sets of goals. Splitting a stakeholder group's viewpoint further into distinct perspectives (and hence distinct agents) corresponding to functional and non-functional goals, the pareto-optimality property of an r-maximal set still holds. In many practical settings, richer and finer-grained, representations exist for agent utility functions suggesting a research road-map in which inconsistency handling in RE is viewed as a process of multi-criteria optimization.

3 SUPPORTING REQUIREMENTS EVOLUTION

We are interested in supporting two fundamental requirements evolution operations. In *addition*, a new requirement is added to a specification. This is the most common kind of evolution operation and becomes necessary when new goals emerge through the process of elicitation, elaboration and refinement. In *removal*, a requirement is removed from a specification. This becomes necessary when a stakeholder group decides to withdraw a goal that it might have earlier (and possibly implicitly) asserted. It is also useful when it becomes apparent in the RE process that certain goals are unachievable. Since specifications are often large, it would be useful to support a removal operation that does not require us to check first to determine if a goal has been explicitly stated (or is logically entailed by explicitly stated goals) before proceeding to remove it.

In this section, we shall define the following two operations: addition of essential requirements and the general removal operation. We observe that the addition operation for tentative requirements is trivial - it simply involves adding the new requirement to the corresponding set of tentative requirements (i.e., T_{FR} or T_{NFR}). We also observe that while we have not considered the question of revising domain theories, this can be achieved with minor modifications to the machinery described below. In our formulation, the input in an addition operation is a justified requirement (i.e., a goal together with its justification) while the input in a removal operation is simply an assertion to be removed. We are interested in the most general versions of these operations, which incorporate a trade-off analysis component to weigh the cost of discarding requirements to accommodate the input (should this become necessary) against the cost of ignoring the change step (such operations are referred to as *non-prioritized belief revision* in the AI literature [10]). Thus, addition and removal operations may fail in our framework as a consequence of the trade-off analysis. We shall define a generic requirements evolution operator as follows:

$$E : \mathcal{R} \times \mathcal{RC} \times \mathcal{OC} \times \mathcal{OP} \times \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{R} \times \mathcal{M}$$

where $\mathcal{R}$ is the class of specifications, $\mathcal{M}$ is the class of r-maximal sets, $\mathcal{OP}$= $\{FR - addition, FR - removal, NFR - addition, NFR - removal\}$, $\mathcal{L}$ is the first-order language in which requirements and their justifications are represented, $\mathcal{RC}$ is the class of *revision choice functions* (to be defined below). and $\mathcal{OC}$ is the class of *outcome choice functions* (as defined earlier). The evolution operator takes as input a specification,

two choice functions, an indication of whether the operation involves addition or removal, a requirement (to be added or removed, depending on the type of operation) and its corresponding justification. As output, an evolution operator produces a revised specification and a preferred r-maximal set of this specification (selected from amongst the possibly many r-maximal sets that may exist).

A *revision choice function* takes as input a set of possible outcomes of an evolution step (where each outcome is denoted by a pair consisting of a set of essential justified functional requirements, and a set of essential justified non-functional requirements) and provides as output an element of this set. Intuitively, a revision choice function selects one of the multiple possible candidate outcome specifications of a requirements evolution operation. Formally, a *revision choice function* c_r is defined as:

$$c_r : 2^{\mathcal{RE}} \rightarrow \mathcal{RE}$$

where $\mathcal{RE}$ is the class of possible (partial) specifications of the form (EF, EN) where EF is a set of justified functional requirements and EN is a set of justified non-functional requirements. The revision choice function encodes all of the criteria used in the trade-off analysis to decide whether a given input is to be accepted, and is thus a critical element of the evolution process.

In an addition operation, the intent is to incorporate the input (essential) justified requirement into the appropriate set of essential requirements (E_{FR} or E_{NFR}) in a manner that causes minimal change to the existing specification and results in a consistent specification. The basic steps are as follows: First, we identify the maximal consistent subsets of a set consisting of the essential requirements of the current specification together with the input requirement. These maximal consistent subsets are somewhat different from r-maximal sets defined in the previous section, since these are generated from a set of justified requirements (as opposed to a full requirements specification). We use the *cons* operator defined below for this purpose. Second, since multiple such maximal consistent subsets might exist, we apply a revision choice function to select one. Note that the selected maximal consistent subset might not include the input requirement that we sought to add, if the process of trade-off analysis (as implemented in the revision choice function) results in a decision to not accept the input (i.e., not include it in the revised set of essential requirements). Third, the selected maximal consistent subset denotes the new set of essential requirements for the revised specification (appropriately partitioned to obtain E_{FR} and E_{NFR}). Fourth, those elements of the original E_{FR} and E_{NFR} that are not included in their revised versions are added to the original sets T_{FR} and T_{NFR} respectively to obtain their revised versions. In other words, prior essential requirements that are not included in the revised set of essential requirements are demoted and retained as tentative requirements. Finally, if the input requirement is rejected (i.e., it does not appear in the revised set of essential requirements), it too is retained as a tentative requirement.

In a removal operation, the intent is to ensure that a given assertion a is not entailed by any of the r-maximal sets of the revised specification. This can be achieved by adding a new essential justified requirement $\emptyset : \neg a$. Notice that the definition of r-maximal sets ensures that every maximal consistent subset is consistent with $\neg a$. In other words, we have a guarantee that the goals together with their justifications do not entail a in any r-maximal set. In principle, we can add $\emptyset : \neg a$ to either E_{FR} or E_{NFR}, but we will assume a convention in which the choice is determined by the kind of goals (i.e., functional or non-functional) that a refers to. The basic steps are as follows: First, we identify maximal consistent subsets of the set consisting of the essential requirements of the current specification together with the requirement $\emptyset : \neg a$. Once again, we use the *cons* operator defined below for this purpose. Second, we use a revision choice function to select one of the possibly many maximal consistent subsets that might exist. Once again, the requirement $\emptyset : \neg a$ might not exist in the selected subset, in the event that the trade-off analysis results in a decision to reject the input. Third, as with addition, we generate the revised sets of essential requirements from the selected maximal consistent subset. Prior essential requirements that are not retained in the revised sets of essential requirements are demoted and retained as tentative requirements. If the input is rejected, it too is retained as a tentative requirement.

The *cons* operator is formally defined as follows: Let FR be any set of justified FRs and NFR be any set of set of justified NFRs. Then:
$cons((FR, NFR)) = \{(FR', NFR') \mid FR' \cup NFR'$ is r-consistent, for any FR'' such that $FR' \subset FR'' \subseteq FR$, $FR'' \cup NFR'$ is not r-consistent and for any NFR'' such that $NFR' \subset NFR'' \subseteq NFR$, $FR' \cup NFR''$ is not r-consistent$\}$.

We now present the formal definition of the equirements evolution operator E. Let E be defined such that:
$E(\Delta, c_r, c_o, op, f, j) = (\Delta', s)$
where:
$\Delta = (D, E_{FR}, T_{FR}, E_{NFR}, T_{NFR})$ is the initial specification.
$\Delta' = (D', E'_{FR}, T'_{FR}, E'_{NFR}, T'_{NFR})$ is the revised specification.
c_r is the input revision choice function.
c_o is the input outcome choice function.
op denotes the specific operation under consideration and must be one of the following: FR-addition, NFR-addition, FR-removal, NFR-removal.

f and j are sentences denoting the requirement and its justification respectively.
s is the preferred r-maximal set of the revised specification.

Each of the four classes of operations are considered separately.

- If op =FR-addition then:
  $(E'_{FR}, E'_{NFR}) = c_r(cons((E_{FR} \cup \{f : j\}, E_{NFR})))$
  $T'_{FR} = T_{FR} \cup ((E_{FR} \cup \{f : j\}) - E'_{FR})$
  $T_{NFR} = T_{NFR} \cup (E_{NFR} - E'_{NFR})$

- If op =NFR-addition then:
  $(E'_{FR}, E'_{NFR}) = c_r(cons((E_{FR}, E_{NFR} \cup \{f : j\})))$
  $T_{NFR} = T_{NFR} \cup ((E_{NFR} \cup \{f : j\}) - E'_{NFR})$
  $T_{FR} = T_{FR} \cup (E_{FR} - E'_{FR})$

- If op =FR-removal then:
  $(E'_{FR}, E'_{NFR}) = c_r(cons((E_{FR} \cup \{\emptyset : \neg f\}, E_{NFR})))$
  $T'_{FR} = T_{FR} \cup ((E_{FR} \cup \{\emptyset : \neg f\}) - E'_{FR})$
  $T_{NFR} = T_{NFR} \cup (E_{NFR} - E'_{NFR})$

- If op =NFR-removal then:
  $(E'_{FR}, E'_{NFR}) = c_r(cons((E_{FR}, E_{NFR} \cup \{\emptyset : \neg f\})))$
  $T_{NFR} = T_{NFR} \cup ((E_{NFR} \cup \{\emptyset : \neg f\}) - E'_{NFR})$
  $T_{FR} = T_{FR} \cup (E_{FR} - E'_{FR})$

In each case: $s = c_o(S(\Delta'))$, where $S(\Delta)$ denotes the set of all r-maximal sets of a specification Δ.

Evolution involves mapping from one specification to another in a process that is iterated over the course of the RE exercise. The preferred r-maximal set at the end of the exercise is taken as the final, consistent set of requirements for downstream activities in the life-cycle. Notice that this is a deferred commitment strategy since no commitment to an r-maximal set is made until one becomes necessary. This avoids premature (and possibly flawed) commitment to specific outcomes. This is also a lazy evaluation strategy since it does not require us to generate r-maximal sets at every step (which require consistency checks involving all requirements) but only the outcomes of the *cons* operator (where consistency checks are restricted to a potentially smaller set of essential requirements).

4 THE REFORM SYSTEM

The REFORM system seeks to support the inconsistency handling and requirements evolution processes and implements many of the strategies discussed in this paper. The system consists of the following key modules:

User interface: The primary requirement for a user interface is that it must offer constructs for defining requirements that are both practitioner-accessible and semantically well-grounded. It must also provide the means to support traceability between the user-level specifications and assertions in an underlying formal language. The KAOS language seeks to do this by offering a user-level language based on semantic nets in addition to a formal assertion layer. The current implementation of the REFORM system supports informal definitions in natural language at the user-level and provides an environment that supports the generation of formal assertions from these informal definitions. It does this by permitting pre-defined (and domain-specific) ontologies to be plugged in. The user is then able to compose formal assertions for each informal definition by using elements of the concept hierarchies in these ontologies, while the corresponding formal concept definitions (also obtained from the plugged-in ontologies) are added to the domain theory. In the spirit of the KAOS language, we plan to extend the user interface with a semantic net-style graphical requirements definition language, specifically one based on conceptual graphs.

Requirements repository: This is a persistent store that maintains a requirements specification. For user-specified goals, it also maintains a set of informal natural language definitions that can be traced back to sets of formal assertions that appear as either essential or tentative requirements (note that this distinction is determined by users). The repository is updated by the revision module.

Theorem prover: This performs the dual functions of satisfiability checking and query answering. The current implementation relies on a basic first-order theorem-prover, but plans for future work include integrating a temporal logic prover to handle the full power of the KAOS formal language.

Consistency management module: This module works with a static snap-shot of the requirements repository, and supports the following three kinds of operations:

- Generating r-maximal sets of the current specification. This is a computationally expensive operation (although some strategies for speeding up this operation are discussed below). These subsets are presented to users as sets of explicitly stated formal assertions (i.e., it does not compute any logical consequences of the explicitly stated goals at this stage).

- Given a r-maximal set that the system has already generated and a query in the form of a formal assertion, the system is able to determine if the query is a logical consequence of the r-maximal set.

- Applying the outcome choice function to the set of all r-maximal sets of a specification (these are assumed to have been explicitly generated). In the full version of this paper, we define an algorithm that interleaves the computation of r-maximal sets with the application of user choice to determine the preferred r-maximal set in a manner that obviates

the need to generate all such sets a priori to perform outcome choice,

Revision module: This module updates the requirements repository using the evolution operator defined earlier. This module involves the application of the revision choice function in a manner similar to the application of the outcome choice function in the consistency management module. Once again, it is possible to use the algorithm referred to above to generate the preferred revision choice outcome via the interactive application of user determined revision choice criteria at run-time.

Given the complexity of generating r-maximal and min-conflict sets, the REFORM system adopts several heuristic strategies. One such strategy involves partitioning the specification such that the set of predicate symbols in each partition is disjoint. Then the scope of each consistency check can be limited to each partition. When an input in a change step can be added to an existing partition without violating the partitioning constraint, then the scope of the consistency check can be limited to the partition that the input belongs to. When an input assertion in a change step has a signature that intersects with the signatures of multiple partitions, then these partitions are merged into a single partition (via a simple set union operation). To ensure that partitions are not needlessly conflated, the input assertion in a change step can be converted to CNF, and a change step performed separately for each conjunct in the resulting formula. Several efficient special cases for the evolution operator exist such as one where restricting all essential requirements and the input to Horn clauses guarantees that an outcome can be generated in polynomial time. More details on these and other strategies are available in [9].

REFERENCES

[1] C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[2] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th Int'l Conference on Software Engineering*, pages 158–165, 1991.

[3] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[4] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh. Coordinating distributed viewpoints: The anatomy of a consistency check. In *Concurrent Engineering: Research and Applications*, CERA Institute, West Bloomfield, USA, 1994.

[5] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8), 1994.

[6] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, 1992.

[7] A. K. Ghose. A formal basis for consistency, evolution and rationale management in requirements engineering. In *Proceedings of the 1999 IEEE International Conference on Tools for AI*, 1999.

[8] A. K. Ghose. Managing requirements evolution: Formal support for functional and non-functional requirements. In *Proceedings of the 1999 International Workshop on Principles of Software Evolution*, pages 118–124, Fukuoka, Japan, 1999.

[9] A. K. Ghose. Managing requirements evolution. Technical report, University of Wollongong, Decision Systems Lab, 2000.

[10] S. O. Hansson. *Belief Base Dynamics*. Uppsala, 1991.

[11] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[12] A. Hunter and B. Nuseibeh. Analyzing inconsistent specifications. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 78–86, 1997.

[13] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[14] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10), 1994.

[15] B. Nuseibeh and A. Russo. Using abduction to evolve inconsistent requirements specifications. *Australian Journal of Information Systems*, 7, 1999.

[16] B. A. Nuseibeh and S. M. Easterbrook. The process of inconsistency management : a framework for understanding. In *Proceedings of the First International Workshop on the Requirements Engineering Process (REP'99)*, Florence, Italy, 1999.

[17] D. Poole, R. Goebel, and R. Aleliunas. Theorist: a logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 331–352. Springer Verlag, 1987.

[18] M. D. Ryan. Defaults in specifications. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 142–149, 1993.

[19] J. J.-P. Tsai, T. Weigert, and H.-C. Jang. A hybrid knowledge representation as a basis for requirement specification and specification analysis. *IEEE Transactions on Software Engineering*, 18(2):1076–1099.

[20] A. van Lamsweerde, R. Darimont, and E. Leiter. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. on Software Engineering*, 24(11):908–926, 1998.

[21] V. Wiels and S. M. Easterbrook. Management of evolving specifications using category theory. In *Proceedings of the 13th International Conference on Automated Software Engineering*, 1998.

[22] D. Zowghi, A. K. Ghose, and P. Peppas. A framework for reasoning about requirements evolution. In *Proc. of the 1996 Pacific Rim Int'l Conf. on AI*, 1996.

5 APPENDIX
EXAMPLE

The following example builds on the TRMCS system case study adopted for the current IWSSD. We present the example in three parts. The first part presents some plausible system requirements and identifies various categories of inconsistency inherent in these assertions. The second part shows how the techniques proposed in this paper can be used to resolve these inconsistencies. The third part demonstrates the use of the requirements evolution operator in this setting. We use a many-sorted first-order language similar to that used in the formal assertion layer in KAOS [20] [3]. As with the KAOS formal language, the requirement names have a temporal flavour (*Achieve*, *Maintain* etc.) but we ommit temporal operators from this example for simplicity.

Part 1: Paramedical professionals (paramedics) and quality assurance (QA) professionals are stakeholders in the system. QA professionals require access to paramedic activity logs to better monitor their performance.Formally:

Goal *Maintain*[QAAccessParamedicActivityLog]
FormalDef ∀ p: Paramedic, q: QAProfessional, l:Activitylog
Records(p,l) → Accesses(q,l)
InformalDef *Activity logs of every paramedic are accessible to all QA professionals.*

Paramedics would require that QA professionals not be given access to activity logs, preferring to restrict access to only to medical practitioners (who might need a record of paramedic interventions to make treatment decisions). Formally:

Goal *Maintain*[ParamedicActivityLogAccess]
FormalDef ∀ p: Paramedic, q: QAProfessional, m: MedicalPractitioner, l:Activitylog
Records(p,l) → Accesses(m,l) ∧¬Accesses(q,l)
InformalDef *Activity logs of every paramedic are accessible to all medical professionals but not to any QA professional.*

Given knowledge of the fact that states of the system exist where:

∃ p: Paramedic, l: ActivityLog
Records(p,l)

becomes true, it is easy to see that the two goals defined above are inconsistent. This is an example of a conflict between distinct stakeholder groups and between distinct functional requirements.

Consider the following goal which requires dispatchers (who dispatch ambulances in response to medical emergencies) to have access to medical records of the patient involved for the duration of the emergency.

Goal *Achieve*[DispatcherAccessPatientRecords]
FormalDef ∀ p: Patient, d: Dispatcher, r: PatientRecord, e: Event
Emergency(e,p) ∧ History(p,r) ∧ Manages(d,e) → AccessesDuringEvent(d,r,e)
InformalDef *If a dispatcher is involved in the management of a medical emergency concerning a given patient, then the dispatcher has access to the medical history of that patient for the duration of the emergency.*

The rationale for this goal is another goal which requires that dispatchers be able to communicate relevant portions of a patient's medical history to paramedics during a medical emergency involving that patient.

Goal
Achieve[PatientRecCommunicatedParamedics]
FormalDef ∀ p: Patient, d: Dispatcher, r: PatientRecord, e: Event, m: Paramedic
Emergency(e,p) ∧ History(p,r) ∧ Manages(d,e) ∧ Responds(m,e) → CommunicatesDuringEvent(d,m,r,e)
InformalDef *Dispatchers managing a medical emergency involving a patient communicate that patient's medical history to paramedics responding to the emergency.*

A different goal requires that mobile computing devices (such as handheld devices used by paramedics, or onboard devices on ambulances) be equipped to directly access patient records from help center data servers. Formally:

Goal *Maintain*[MobileAccessPatientRecords]
FormalDef ∀ c: MobileComputingDevice, r: PatientRecord
DeviceAccess(c,r)

The rationale for this is a goal which requires that paramedics be able to directly access a patient's medical records during an emergency involving that patient. Formally:

Goal *Achieve*[ParamedicAccessPatientRecords]
FormalDef ∀ p: Patient, r: PatientRecord, e: Event, m: Paramedic
Emergency(e,p) ∧ History(p,r) ∧ Responds(m,e) → AccessesDuringEvent(m,r,e)
InformalDef *Paramedics responding to a medical emergency involving a patient are able to directly access that patient's medical history.*

Consider the following goal which seeks to minimize threats to security and privacy by avoiding redundant access to patient records. Formally:

Goal *Avoid*[RedundantAccess]
FormalDef ∀ x,y: HealthProfessional, r: PatientRecord, e: Event
CommunicatesDuringEvent(x,y,r,e) ∧ x≠y → ¬AccessesDuringEvent(y,r,e)
InformalDef *If a patient history is communicated to a health professional y by another health professional x during an event, then y does not require direct access to the patient history during that event.*

Note that both dispatchers and paramedics belong to the HealthProfessional sort. Given knowledge of the fact that states of the system exist where:

∃ p: Patient, d: Dispatcher, r: PatientRecord, e: Event, m: Paramedic
Emergency(e,p) ∧ History(p,r) ∧ Manages(d,e) ∧ Responds(m,e)

we are able to detect that the goals *Achieve*[PatientRecCommunicatedParamedics], *Achieve*[ParamedicAccessPatientRecords] and *Avoid*[RedundantAccess] are jointly inconsistent. The goals *Achieve*[DispatcherAccessPatientRecords], *Maintain*[MobileAccessPatientRecords] and *Avoid*[RedundantAccess] are also potentially inconsistent, since the rationale for the first two goals contradict the third goal. Intuitively, a requirement to provide dispatchers access to patient records (in order that they may communicate these to paramedics during an emergency) contradicts a requirement to equip mobile computing devices with access capability to the patient record database (in order that paramedics might directly access such records during an emergency) if we also wish minimize redundant access privileges. This shows that goals may sometimes conflict because their rationale contradict each other.

Consider the non-functional requirement that we maintain fast access to patient records. Formally:

Goal *Maintain*[FastAccessPatientRecords]
FormalDef ∀ u: User, r: PatientRecord, t: TimeInterval
AccessDelay(u,r,t) → t≤30
InformalDef *The delay in accessing a patient record must be no more than 30 seconds.*

Consider another (high-level) non-functional goal *Maintain*[PatientPrivacy]. This is refined (possibly by an AND-refinement link) to obtain the following functional goal, amongst others:

Goal *Maintain*[SecureAccessPatientRecords]
FormalDef ∀ u: User, r: PatientRecord
AccessRequest(u,r) → Authenticate(u,r)
InformalDef *If a user requests access to a patient record, then the system must authenticate that request.*

Consider a domain theory that indicates that authentication of an access request will necessarily make the delay in accessing a patient record greater than 30 seconds. Formally:

∀ u: User, r: PatientRecord, t: TimeInterval
Authenticate(u,r) ∧ AccessDelay(u,r,t) → t>30

Given this domain theory (as well as knowledge that system state exists where ∃ u: User, r: PatientRecord AccessRequest(u,r) is true), the goals *Maintain*[FastAccessPatientRecords] and *Maintain*[PatientPrivacy] are inconsistent. So are *Maintain*[FastAccessPatientRecords] and *Maintain*[SecureAccessPatientRecords]. The former is an example of a conflict between two non-functional requirements while the latter is an example where a functional requirement contradicts a non-functional requirement.

Part 2: In this section, we shall demonstrate the use of the inconsistency handling techniques defined in this paper on the example presented above. For brevity, we shall assume that goal names stand for their (equivalent) formal definitions. We first construct a requirements specification S as follows:

- D_{inv} = {∀ u: User, r: PatientRecord, t: TimeInterval Authenticate(u,r) ∧ AccessDelay(u,r,t) → t>30}
- D_{traj} = {[{∃ p: Paramedic, l: ActivityLogRecords(p,l), ∃ u: User, r: PatientRecord AccessRequest(u,r), ∃ p: Patient, d: Dispatcher, r: PatientRecord, e: Event, m: Paramedic Emergency(e,p) ∧ History(p,r) ∧ Manages(d,e) ∧ Responds(m,e) }]}
- E_{FR} = {*Maintain*[QAAccessParamedicActivityLog]:∅}
- T_{FR} = {*Achieve*[DispatcherAccessPatientRecords]:

Achieve[PatientRecCommunicatedParamedics], *Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords], *Maintain*[SecureAccessPatientRecords]:$\emptyset$}

- $E_{NFR} = \{$*Avoid*[RedundantAccess]:$\emptyset\}$
- $T_{NFR} =$ {*Maintain*[FastAccessPatientRecords]:$\emptyset\}$

Two min-conflict sets can be generated from this specification:

- {*Achieve*[DispatcherAccessPatientRecords]: *Achieve*[PatientRecCommunicatedParamedics], *Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords]}
- {*Maintain*[SecureAccessPatientRecords]:$\emptyset$, *Maintain*[FastAccessPatientRecords]:$\emptyset\}$

Four r-maximal sets can be generated from this specification:

- $E_{FR} \cup E_{NFR} \cup$ {*Achieve*[DispatcherAccessPatientRecords]: *Achieve*[PatientRecCommunicatedParamedics], *Maintain*[SecureAccessPatientRecords]:$\emptyset\}$
- $E_{FR} \cup E_{NFR} \cup$ {*Achieve*[DispatcherAccessPatientRecords]: *Achieve*[PatientRecCommunicatedParamedics], *Maintain*[FastAccessPatientRecords]:$\emptyset\}$
- $E_{FR} \cup E_{NFR} \cup$ {*Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords], *Maintain*[SecureAccessPatientRecords]:$\emptyset\}$
- $E_{FR} \cup E_{NFR} \cup$ {*Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords], *Maintain*[FastAccessPatientRecords]:$\emptyset\}$

Part 3: This section provides examples of the use of the evolution operator defined in this paper. We shall consider the addition operation first, by adding the functional requirement *Maintain*[ParamedicActivityLogAccess]: $\emptyset$ to the specification S defined above. The *cons* operator generates the following two outcomes:

- ({*Maintain*[QAAccessParamedicActivityLog]:$\emptyset\}$, E_{NFR})
- ({*Maintain*[ParamedicActivityLogAccess]: $\emptyset\}$, E_{NFR})

A revision choice function c_r is applied to these outcomes which incorporates the trade-off analysis required to decide whether the input is to be accepted.

If the first outcome is chosen, we obtain $S' =< (D_{inv}, D_{traj}), E_{FR}, T_{FR} \cup \{$*Maintain*[ParamedicActivityLogAccess]: $\emptyset\}, E_{NFR}, T_{NFR} >$ as the revised specification. Notice that this outcome corresponds to a decision to reject the input (which is retained as a tentative requirement).

If the second outcome is chosen, we obtain $S'^{*} =< (D_{inv}, D_{traj}),$ {*Maintain*[ParamedicActivityLogAccess]:$\emptyset\}, T_{FR} \cup$ {*Maintain*[QAAccessParamedicActivityLog]:$\emptyset\}$, $E_{NFR}, T_{NFR} >$ as the revised specification. Notice that this outcome corresponds to a decision to retain the input as an essential requirement, while demoting the prior essential requirement (which conflicts with the input) to the status of a tentative requirement.

Assume that we select S' as the revised specification. We now wish to remove the goal *Achieve*[PatientRecCommunicatedParamedics] from S' (possibly because of privacy concerns arising from insecure communication). The *cons* operator generates a unique outcome (i.e., the input does not contradict any of the current essential requirements) leading to a unique revised specification $S' =< (D_{inv}, D_{traj}), E_{FR} \cup \{\emptyset : \neg$*Achieve*[PatientRecCommunicatedParamedics]$\}$, $T_{FR} \cup \{$*Maintain*[ParamedicActivityLogAccess]: $\emptyset\}, E_{NFR}, T_{NFR} >$

Two r-maximal sets can be generated from S':

- $E_{FR} \cup E_{NFR} \cup$ {*Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords], *Maintain*[SecureAccessPatientRecords]:$\emptyset\}$
- $E_{FR} \cup E_{NFR} \cup$ {*Maintain*[MobileAccessPatientRecords]: *Achieve*[ParamedicAccessPatientRecords], *Maintain*[FastAccessPatientRecords]:$\emptyset\}$

Notice that the goal *Achieve*[DispatcherAccessPatientRecords]: *Achieve*[PatientRecCommunicatedParamedics] does not appear in any of these r-maximal sets since the rationale for the goal has been retracted.