An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture

Mark Klein
Applied Research Lab, Pennsylvania State University, State College PA 16804-0030

Abstract

This paper describes C-ReCS, a WWW-based system for collaborative requirements capture that uses a generic knowledge-based approach to detecting and resolving the "exceptions" (i.e. different kinds of inconsistency, incompleteness and incorrectness) that can arise frequently in such settings.

1 The Challenge

The current DoD requirements definition process typically consists of multiple iterations of increasingly broader reviews, centered around an action officer, with flow times of 400 days or even more. The process thus operates like traditional "throw over the wall" manufacturing, where one function completes its job and only then sends it to the next function for review, with all the attendant problems of excessive iteration and rework, communication delays, misunderstandings between different functions and limited concurrency. While details differ, this kind of serial requirements definition process is also characteristic of much of industry.

Concurrent engineering ideas, which have proven highly successful in manufacturing settings, also appear applicable to the requirements definition process. This approach involves flattening a serial cycle into concurrent activity by a members of a multi-functional team. Team members can get quick feedback from other functions as they explore alternatives, as opposed to only after they have created a complete requirements definition document. This greatly reduces the time and cost needed to produce a final satisfactory requirements set.

A critical challenge to the effectiveness of multi-functional teams, however, is the frequency and impact of coordination problems like inconsistency (for example from differing terminology), incompleteness (for example from dropped balls, unclear responsibilities) and incorrectness (for example from one function creating requirements that prove unviable from the perspective of another function. These "exceptions" often require a tedious and resource-intensive process to detect them, but can have a severe impact on cost, schedule and quality if left unaddressed for too long. This paper describes how one can provide computer support for anticipating/detecting exceptions as well as helping stakeholders to resolve them effectively.

2 Contributions and Limitations of Existing Work

While computer systems to support requirements capture do exist (e.g. The Requirements Manager from Cimflex Teknowledge (5) and RDD-100 from Ascent Logic Inc.), they are designed for single users. One of the key lessons of empirical studies of requirements capture, however, is that the inputs of multiple stakeholders need to be integrated to produce a satisfactory requirements document (18). Existing work on collaborative requirements capture has focused largely on defining normative requirements capture processes (17) or on social rather than substantive elements of the interaction between team members (e.g. 24; 19). C-ReCS focuses by contrast on identifying miscoordination-caused problems with the content of the requirements themselves and then intervening to resolve these problems. There has been a relatively large body of work on dealing with the subclass of exceptions known as design conflicts (see, for example, the September 1994 CERA Journal Special Issue on Conflict Management in Concurrent Engineering, Volume II, Number 3) as well as some preliminary work on requirements capture (e.g. 23; 3; 8). C-ReCS, however, the first system to provide a comprehensive approach to detecting and resolving the full range of collaborative requirements capture exceptions.

3 The C-ReCS System

C-ReCS (the Collaborative Requirements Capture System) is a computer tool for supporting collaborative capture of design decisions, in particular from the early (i.e. requirements definition) phases of the product life cycle. Its functional architecture consists of a centralized

shared requirements database accessed by standard World Wide Web browsers plus exception management services. C-Recs thus supports "virtual" multi-functional requirements definition teams. Anyone who needs to add to or critique a set of requirements can do so using a Web browser at a time and place convenient for them, concurrently with other users.

Requirements and their rationale are captured using a formal language that integrates previous work on requirements (15; 21) and rationale (13; 12) representations. The C-ReCS language, in brief, captures requirements and their rationale as a semantic net consisting of pre-defined entity (e.g. module, interface, resource, attribute) and relation (e.g. is-supported-by, has-submodule, has-value) types. The C-ReCS language is too large to cover adequately here and will be described in detail in a companion paper. This paper will focus on the C-ReCS exception management services, which consist of three major functional components:

- A suite of exception detecting tools that attempt to anticipate/detect symptoms of underlying consistency, completeness and correctness problems in the requirements database.

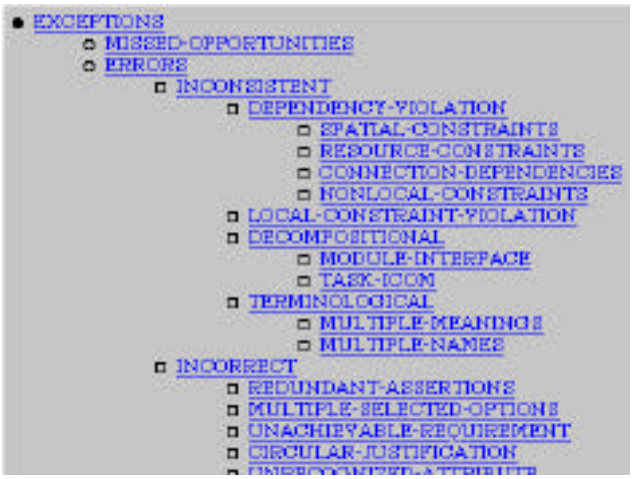
- A diagnosis procedure that attempts to identify one or more candidate diagnosis (i.e. underlying causes) for detected exceptions by examining a knowledge base of generic requirements problem diagnoses.

- A resolution procedure that attempts to identify one or more potential resolutions for the diagnosed problems by examining a knowledge base of generic exception resolution strategies associated with the diagnoses knowledge base.

C-ReCS is currently implemented in Common Lisp using the cl-http server developed at MIT (20), and builds upon previous systems developed by the author for design conflict management (11; 14), rationale capture (13) and workflow management (4). The following sections will describe the exception detection, diagnosis and resolution components of C-ReCS in turn, followed by an example of C-ReCS in operation.

3.1 Exception Detection

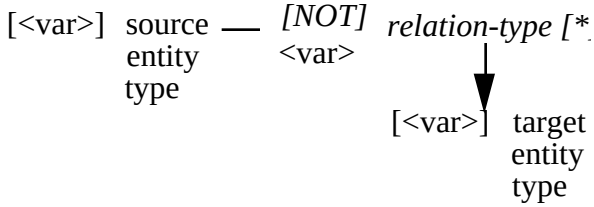
Exceptions can manifest themselves in many ways, and can be usefully organized into a taxonomy of exception types. Some of the types currently supported by C-ReCS include:



Each type of exception manifestation has associated methods for detecting it. While details differ, all the methods are instantiations of three basic mechanisms - graph matching, constraint checking and case-based comparison - described in the sections below.

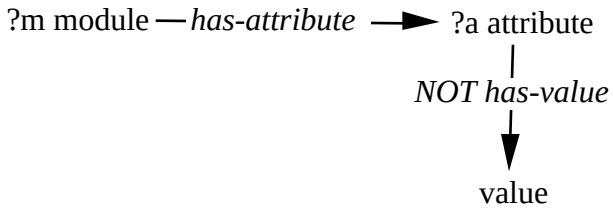
Graph Matching: This approach detects exceptions by looking for characteristic patterns in the semantic net representing the current set of requirements. It is well-suited for detecting incompleteness exceptions and as a building block for detecting other kinds of exceptions. This approach works by specifying a pattern to search for and then potentially applying some additional predicates to the retrieved patterns to determine if they represent exceptions.

Patterns can be specified using a simple language built up of the following primitive elements:

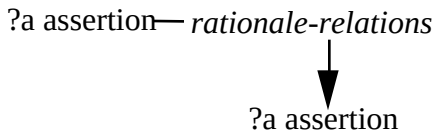


Each such element matches a relationship of a given type between two entities of the given types. If the relation-type is preceded by a "NOT", then the pattern searches for an entity of type *source-entity-type* on which no relations of the given type exist. If the relation-type is followed by a *, then the pattern is matched if one or more links of the given type exist between the source and target . Entities and relations can be preceded by variables (with the format ?<string>) that are bound to the matching entities and relations. These bindings can be passed to predicates to do further checking. More complex patterns can be built up by concatenations of these basic elements.

If, for example, we want to look for a "dangling" module attribute (i.e. one for which no requirements have been specified) we could use the following pattern:



The result will be a list of modules and associated attributes. If we want to look for assertions with circular justifications (i.e. that directly or indirectly justify themselves), we can use the following simple pattern:



where "rationale-relations" is the class of all rationale relations like "supports", "denies" etc.

Constraint Consistency Checking: This approach detects exceptions by looking for unsatisfiable (i.e. inconsistent) sets of decisions concerning a given design parameter. It is well-suited to detecting violations of both absolute constraints (e.g. datatype and range constraints on a given parameter) as well as dependencies across different design parameters. In the former case, the constraints on the parameter are combined with each other to produce their simplest expression, which is then examined to see if it is satisfiable. For example, if we have the constraints (from 1 to 100) and (from 200 to 300) on a design parameter, they can be combined straightforwardly to infer that they can't both be satisfied. Violations of dependencies across multiple design parameters are uncovered using a constraint propagation approach. A commitment for a given design parameter is "propagated" through a relational constraint representing that dependency to produce an inferred constraint on a related design parameter. If one parameter is required to be twice the value of another, for example, then the constraint (> 4) on the first parameter becomes (> 8) when propagated to the second parameter. Once all the constraints on a given design parameter have been determined by this propagation step, we can use the constraint consistency checking methods described above to determine if a conflict exception has occurred.

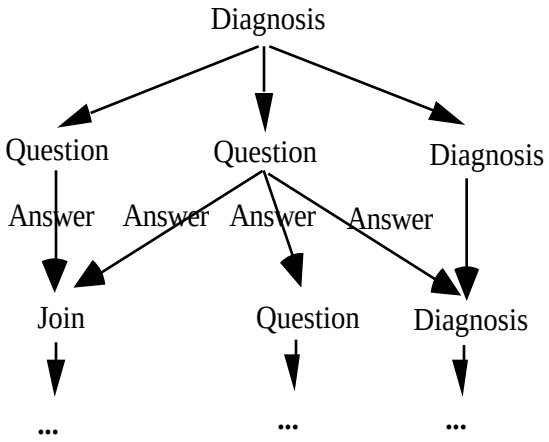
Case-Based Comparisons: This approach detects exceptions by comparing the current requirements set with similar cases and/or with generic requirements templates appropriate for the system of interest (23). It is useful for finding "incorrectness" type exceptions wherein one checks, for example, if a requirement (e.g. maximum speed for a land vehicle) has a value radically different than that in similar cases or applicable templates. Two key components of this approach are a library of cases and/or templates, as well as a retrieval mechanism that is effective at finding applicable cases. The case retrieval mechanism must incorporate a "similarity" metric that accounts for attribute value differences, relative importance of different attributes in calculating the similarity value, and perhaps even the structural similarity of the graph structures in the two cases. C-ReCS currently only implements a simple template retrieval mechanism wherein generic templates are indexed by the modules they define requirements for.

It is often not appropriate to do all kinds of exception detection at all times. Checking for completeness of the requirements set in the early phases of requirements definition could, for example, overwhelm the user with a long list of incompletely specified requirements. It would probably be more appropriate to do this kind of checking much later on. Inconsistency and incorrectness detection, however, may be appropriate throughout the requirements capture process. C-ReCS accordingly allows the user to select what kinds of exception manifestations to look for.

3.2 Exception Diagnosis

C-ReCS uses a heuristic classification approach for diagnosing exceptions (2; 7). Potential diagnoses are arranged into a taxonomy ranging from the very abstract at the top to the very specific at the bottom. The diagnosis mechanism works in a top-down way by iteratively increasing the specificity of a diagnosis based on the symptoms, underlying decision rationale and other factors as well. This is essentially a "shallow model" approach (1) because it is based on compiled empirical and heuristic expertise rather than first principles. This approach is appropriate for domains, such as medical diagnosis (and requirements definition as well), where complete and consistent first-principle-based behavioral models do not exist (in contrast, for example, to computer hardware fault diagnosis where such models do exist and fault causes can be inferred deductively; see for example 6 and 10). An important characteristic of heuristic classification is that the diagnoses represent *hypotheses* rather than guaranteed *deductions:* multiple diagnoses may be suggested by the same symptoms, and often the only way to verify a diagnosis is to see if the associated prescriptions are effective.

The diagnosis hierarchy is structured as follows:



Diagnosis works by a decision-tree-like traversal of this structure wherein the system starts at the top most abstract diagnosis and attempts to refine it to more specific diagnoses. Diagnoses have preconditions attached to them that represent the characteristics that distinguish that class from its direct superclass. The complete defining characteristics for a diagnosis will thus be the conditions for that class plus for all of its superclasses. The diagnosis taxonomy also allows one to define "question" nodes, that make some query, followed by "answer" arcs, one or more of which are traversed based on the result of the query. Question nodes and answer arcs are optional but they have proved useful as a way of making the structure of the diagnosis taxonomy more understandable. Multiple alternative paths may lead to the same diagnosis or question, so the default semantics for multiple parents is disjunctive. One can specify conjunctive semantics using a "join" node. Traversal proceeds recursively and can result in more than one candidate diagnosis, since multiple causes may be suggested by the same symptoms. A portion of the diagnosis taxonomy currently used by C-ReCS is given below:

- QUERY underlying cause type?
  - ANSWER inconsistent
    - DIAGNOSIS over-ambitious constraints
      - DIAGNOSIS requirement region too big
      - DIAGNOSIS performance goals too high
        - STRATEGY relax requirements
      - DIAGNOSIS resource needs too great
    - DIAGNOSIS chose wrong alternative
      - DIAGNOSIS chose wrong plan
      - DIAGNOSIS chose wrong component
    - DIAGNOSIS used rough guess
    - DIAGNOSIS didn't fully propagate changes
    - DIAGNOSIS differing optima
    - DIAGNOSIS missing detail
    - DIAGNOSIS user doesn't understand dependencies
    - DIAGNOSIS replicated attribute
      - STRATEGY consolidate attributes
  - ANSWER incorrect
    - DIAGNOSIS over-ambitious constraints
    - DIAGNOSIS chose wrong alternative

3.3 Exception Resolution

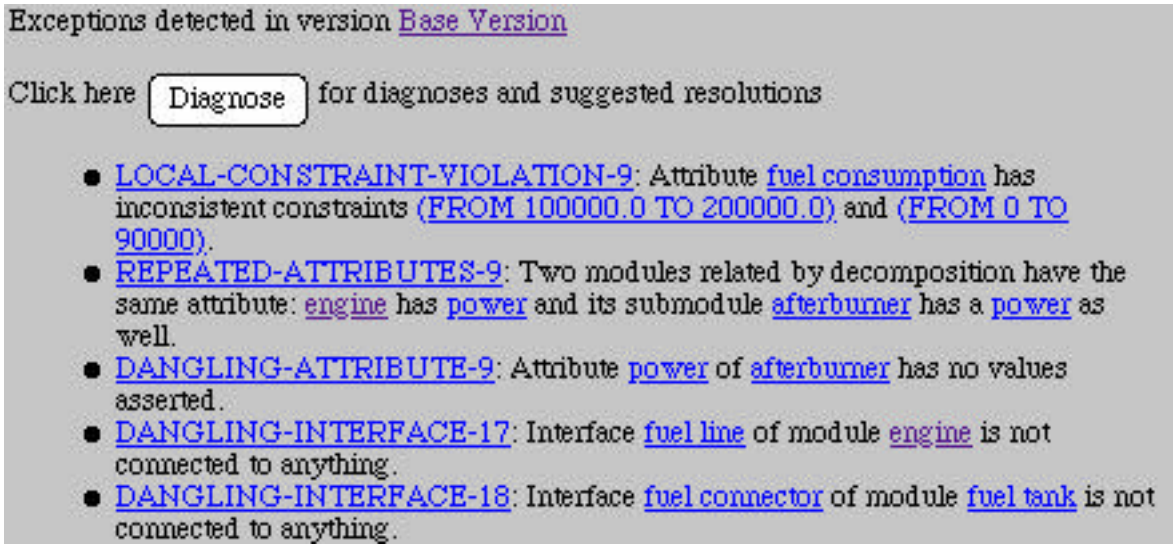
Once one or more candidate diagnoses for an exception have been identified, the next step is for C-ReCS to generate, using its knowledge base of generic exception resolution strategies, specific suggestions for resolving the diagnoses. A diagnosis class will often have several potential resolution strategies available. Since they may not all be applicable for a particular exception, a decision tree procedure identical to that used to select diagnoses is used to find the generic strategies for a given diagnosis. Strategies are represented as natural language text templates with a number of "slots". These slots are filled with context-specific values, found using the query language, to produce specific suggestions.

Imagine for example that C-ReCS has made the diagnosis "Poor plan choice". The associated generic resolution strategy ("Try alternate plan <plan> for goal <goal>") can be instantiated into specific suggestion by finding the goals underlying the exception as well as the untried plans for the goal; we then can instantiate the strategy template into specific suggestions like "Try alternate plan use-hybrid-power-system for goal refine-power-system-design". A suggested resolution strategy once instantiated is annotated with rationale capturing the goal the strategy is designed to achieve.

3.4 An Example

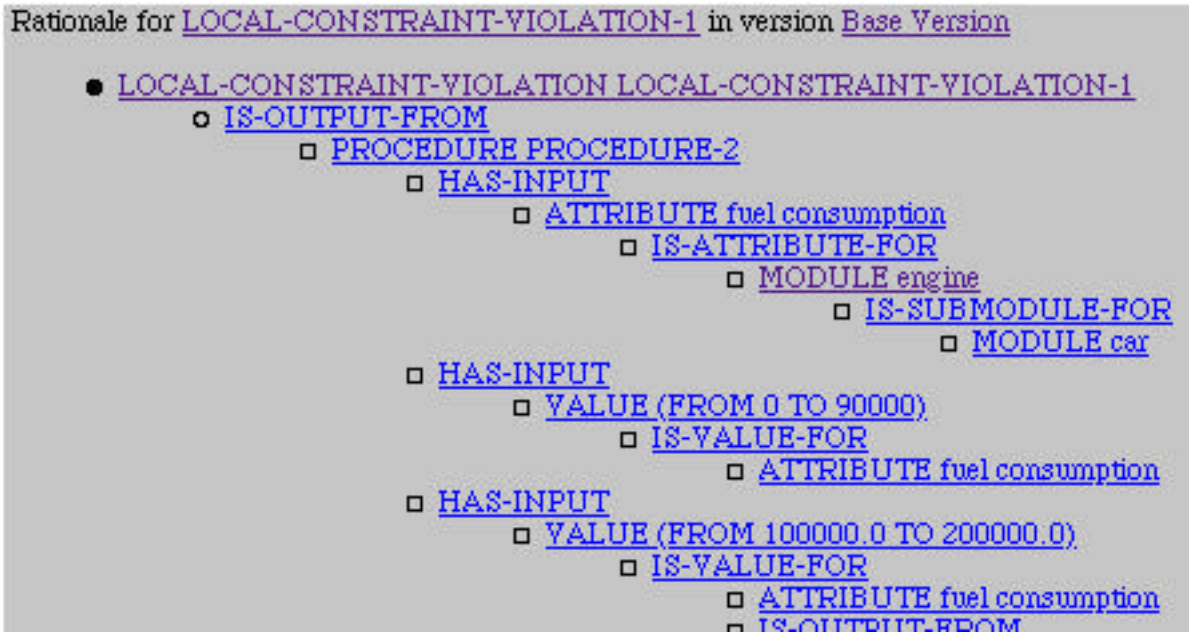
Imagine we are performing some preliminary requirements definition for a car. So far we have specified some of the major components as well as some requirements for these components. The engine, for example, is required to have a fuel line interface as well as prescribed ranges on its fuel consumption and power. A typical efficiency value as well as a mathematical expression describing the relationship between efficiency, fuel consumption and power has also been added:

The C-ReCS user can at any time elect to invoke the exception detection procedure. When invoked, C-ReCS allows users to select the subset of the available exception detection methods they want to use and then presents a summary of the exceptions detected:



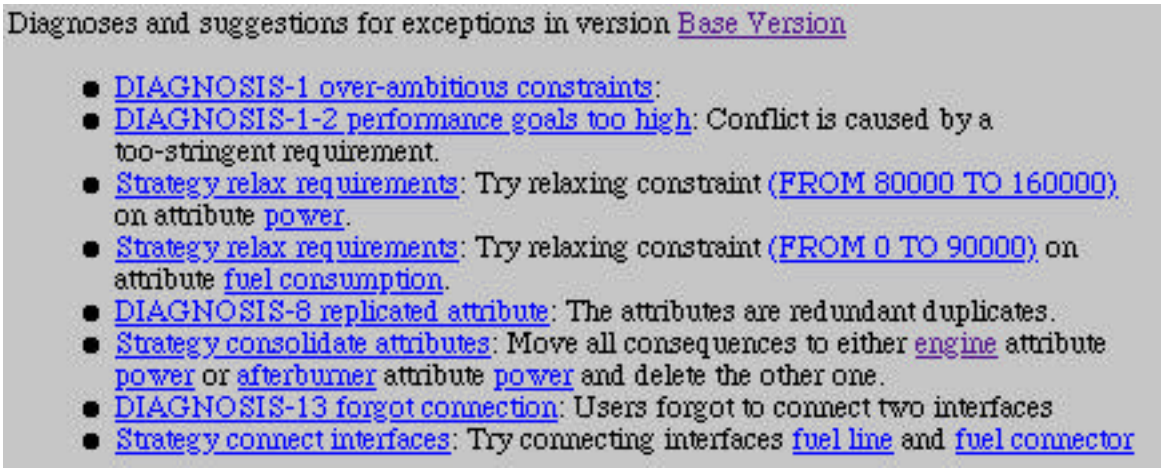
In this case C-ReCS finds, based on propagation of the design constraints, that the requirements on engine fuel consumption are inconsistent. It also notes that the "power" attribute has been placed on both the engine component and its sub-component (a potential inconsistency or incorrectness exception), that there is an attribute with no constraints (an incompleteness exception) and that there are two module interfaces that are not connected to anything (also incompleteness exceptions).

Exception assertions are all annotated automatically with their rationale. The rationale graph for the constraint violation partly reproduced below, for example, reveals (after some effort) the requirements and inference procedures logically supporting that exception:



C-ReCS users can of course attempt to resolve exceptions on their own, but this can be a tedious and error-prone process. Exceptions typically represent symptoms of some underlying problem that may not be obvious on first inspection. A single symptom can derive from many possible causes. Conversely, multiple exception symptoms can be caused by a single underlying problem. One will typically be forced to do extensive browsing of a complex requirements database to fully understand the exceptions, and it is of course possible that one will not hit upon the correct diagnoses and effective resolutions for these diagnoses.

The C-ReCS exception diagnosis and resolution suggestion procedures address this problem by suggesting diagnoses and resolutions for exceptions. In this example, C-ReCS makes the following suggestions:



C-ReCS suggests that the constraint violation conflict is due to over-ambitious design constraints, traces back through the exception rationale to find the two potentially changeable design requirements underlying the conflict, and suggests that one (or both) of them be relaxed. C-ReCS also suggests that the replicated attribute exception is due

to the same attribute appearing in two places by mistake, and suggests that the user consolidate them. Finally, the system hypothesizes that the dangling interfaces exceptions are due to the users forgetting to connect two interfaces, verifies that they are of compatible types, and suggests that they should be connected. The five exceptions in this case appear to have stemmed from three underlying problems. C-ReCS automatically records the rationale for any suggested diagnosis (i.e. in terms of the exceptions it covers) and associated strategies (i.e. in terms of the goal of resolving the diagnosed problem). If a user decides to implement a suggested resolution strategy, any resulting changes to the requirements set will be captured as the outcomes of this strategy so it will be clear for later reference why they were made.

C-ReCS can not guarantee that any given proposed diagnosis or resolution is appropriate. The constraint conflict above, for example, may only be resolvable by selecting a radically different design configuration with greater potential engine efficiency. It's added-value stems, rather, from using its' exception handling expertise to quickly generate a rich set of ideas for the users to consider when dealing with their particular problems. Even if none of the proposed diagnoses or strategies are accepted as is, they often can serve as a starting point for creating effective responses to these problems.

4 Contributions of this Work

The C-ReCS system addresses some central issues in requirements capture. A study of software requirements elicitation in 23 project organizations (16) revealed the key issues to be:

- communication
- agreement about requirements
- managing change

C-ReCS addresses important aspects of all these problems, by providing a tool by which multiple stake-holders can become aware of the affects of their requirements decisions and be guided through a resolution process when problems arise, for both initial requirements decisions as well as changes in original requirements.

Another way to look at the contribution of this work is to follow the classification presented in 22; which distinguishes three orthogonal goals of requirements capture:

- Specification: transform vague set of ideas into a sufficiently detailed & complete system specification
- Representation: transform informally represented knowledge into (one or more integrated) formal representations
- Agreement: transform a set of (possibly contradictory) personal views into a single consensus view

This work can thus be seen as helping to *specify* the requirements in more detail (by reducing incompleteness and incorrectness) as well as foster increased *agreement* among multiple stakeholders (by reducing inconsistency among the requirements they describe). This approach does not address the issue of transforming the representations, but rather assumes as we have noted that the users will use a (quite formal) requirements language.

From a technical perspective, the contribution of this work is that it represents not a single exception management algorithm but rather a generic structure for utilizing an extensible knowledge base of exception management strategies applicable to a wide range of domains. As noted earlier, other work has presented either just a small specialized set of exception handling strategies and/or has been limited to a single domain. This exception management procedure is a generalization of previous work by the author (14; 11) on managing conflicts in collaborative design, and differs from that work in a number of important ways including the inclusion of a much more expressive requirements and rationale capture language, a broader range of exception detection mechanisms, a rationalized structure for the exception cause taxonomy as well as a knowledge base that was significantly extended to handle requirements domain exceptions, not just collaborative design conflicts.

5 Future Challenges

The current system maintains a single database of requirements that is reviewed as a whole when exception detection is invoked. A distributed "lazy" exception management approach (i.e. one based on multiple local databases (i.e. "viewpoints") where inter-viewpoint exceptions are tolerated at least temporarily) (3; 9) is probably appropriate since it provides greater discretion to individual users (they can work on their own for a while before being forced to establish consistency with requirements defined by others), and avoids the potential bottleneck of a centralized database approach (users do not need to constantly monitor and update a sole source repository but can do so at their convenience). This approach raises a number of challenging issues, however, such as when exception detection gets invoked, and how we can best resolve exceptions after a large number of them have accumulated.

The approach described in this paper focuses on repairing faults (an "illness" oriented approach) rather than on disciplines to prevent them from occurring in the first place (a "wellness" approach). Clearly, just as in medicine, both approaches are needed and are in fact complimentary. Some potential wellness approaches include normative requirements capture processes (18), educating users, using shared data dictionaries to foster consensus about domain term semantics (9) etc.

6 References

[1] Chandrasekaran, B. and Mittal, S. Deep Versus Compiled Knowledge Approaches To Diagnostic Problem Solving. *Int. J. Man-Machine Studies* (1983), 425-436.

[2] Clancey, W.J. Classification Problem Solving. *AAAI* (1984), 49-55.

[3] Easterbrook, S. Coordinating Distributed Viewpoints: the anatomy of a consistency check. *Concurrent Engineering Research and Applications: Special Issue on Conflict Management in Concurrent Engineering II*, 3 (September 1994), 209-222.

[4] Faragher-Horwell, R., Klein, M., and Zarley, D., "Overview and Functional Specifications for TCAPS Task Coordination And Planning System: A Computer-Supported Workflow Management System," , Boeing Computer Services Technical Report, no. BCS-G2010-130, December 1992.

[5] Fiksel, J. The Requirements Manager: A Tool for Coordination of Multiple Engineering Disciplines. In *Proceedings of CALS & CE '91*, 1991.

[6] Genesereth, M.R. Diagnosis Using Hierarchical Design Models. In *AAAI-82*, 1982, pp. 278.

[7] Gomez, F. and Chandrasekaran, B. Knowledge Organisation And Distribution For Medical Diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics SMC-11*, 1 (January 1981), 34-42.

[8] Heitmeyer, O., Labaw, B., and Kiskis, D. Consistency Checking of SCR-Style Requirements Specifications. In *Proceedings of the International Symposium on Requirements Engineering*, 1995.

[9] Johnson, W.L. and Harris, D.R. Requirements Analysis Using ARIES: Themes and Examples. In *Proceedings of the Fifth Annual Knowledge-Based Software Assistant Conference*, 1990.

[10] de Kleer, J., Macworth, A.K., and Reiter, R. Characterizing Diagnoses. In *AAAI-90*, AAAI, AAAI, 1990, pp. 324-330.

[11] Klein, M. Supporting Conflict Resolution in Cooperative Design Systems. *IEEE Systems Man and Cybernetics 21*, 6 (December 1991).

[12] Klein, M. DRCS: An Integrated System for Capture of Designs and Their Rationale. In *Proceedings of Second International Conference on Artificial Intelligence in Design*, Pittsburgh, Pennsylvania, 1992.

[13] Klein, M. Capturing Design Rationale in Concurrent Engineering Teams. *IEEE Computer* (January 1993).

[14] Klein, M. Supporting Conflict Management in Cooperative Design Teams. *Journal on Group Decision and Negotiation 2*(1993), 259-278.

[15] Kott, A. and Peasant, J. Representation and Management of Requirements: The RAPID-WS Project. *Concurrent Engineering Research and Applications 3*, 2 (1995), 93-106.

[16] Lubars, M., Potts, C., and R., C.R. Developing Initial OOA Models. In *Proceedings of the International Conference on Software Engineering*, IEEE CS Press, 1993.

[17] Macaulay, L., F., C.H.F., K., M.K., and H., A.F.H. USTM: A New Approach to Requirements Specification. *Interacting with Computers 2*, 1 (1990 1990), 92-117.

[18] Macaulay, L. Requirements Capture as a Cooperative Activity. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, IEEE, 1993.

[19] Macaulay, L., O'Hare, G., Dongha, P., and Viller, S. *Cooperative Requirements Capture: Prototype Evaluation*, John Wiley & Sons (June 1994).

[20] Mallery, J.C. A Common LISP Hypermedia Server. In *Proceedings of the First International Conference on the World Wide Web*, CERN, May 1994.

[21] Nirenburg, I. and Kott, A., "RSL Syntax/Semantics," , Technical Report, no. CDRL A012, Five PPG Place, Pittsburg PA 15222, August 1994.

[22] Pohl, K. The Three Dimensions of Requirements Engineering: A Frameworkand its Applications. *Information Systems 19*, 4 (1994), 234-248.
[23] Sinha, A.P. and Popken, D., "Completeness and Consistency Checking of System Requirements: An Expert Agent Approach," 1995, .
[24] Viller, S. The Group Facilitator: A CSCW Perspective. In *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, 1991.