Supporting Event Based Traceability through High-Level Recognition of Change Events

Jane Cleland-Huang*, Carl K. Chang+, and Yujia Ge+
International Center for Software Engineering, University of Illinois at Chicago
*Now at DePaul University, + Now at Iowa State University
{jhuang@cs.depaul.edu, chang@cs.iastate.edu}

Abstract

Although requirements traceability is crucial in both the development and maintenance of a software system, traceability links and related artifacts tend to deteriorate, as time-pressured practitioners fail to systematically update them in response to change. Event-based traceability addresses this issue by establishing links through a loosely coupled publisher/subscriber scheme. Dependent entities subscribe to requirements and receive event notifications as changes occur. This paper focuses upon the role played by the requirements specification as a publisher of events. A set of standard change events is defined and a method for monitoring a user' s actions within a requirements management environment and the subsequent recognition and publication of the change events is proposed. Early results obtained from testing this approach are reported.

1. Introduction

Requirements traceability, which has been described as "*the ability to follow the life of a requirement in both a forward and backward direction*" [1], is crucial to the successful development and management of all non-trivial software systems. Its significance is illustrated by the fact that the U.S. Department of Defense currently invests about 4 percent of its total IT budget on traceability issues [2]. Unfortunately, despite recognition of its importance, practice has shown that many problems hinder the implementation of a successful and maintainable traceability plan, and ultimately affect the ability of developers to build quality systems on schedule and to manage change effectively. In their 1999 chaos report [3], the Standish group revealed that in the previous year overruns and failed projects cost U.S. companies a staggering $97 billion! Several researchers have found that inadequate traceability is a major contributing factor in such project over-runs [4][5] and its failure limits the ability to manage change effectively [6].

Unfortunately, despite the importance of requirements traceability, practitioners find it extremely difficult to maintain accurate traceability links throughout the life of a software system. Although many requirements management tools such as DOORS and Rational Pro [7] include features that support impact analysis and testing for requirements coverage, they provide little support for guiding the process of creating new links and resolving existing ones when changes are introduced. Brian Azelborn, a fifteen year veteran of Rockwell Collins highlighted this problem by stating that due to time-pressures "it is often the case that relationships are not created", and that when changes occur "existing relationships are not maintained" [8].

Event Based Traceability (EBT), addresses this problem by providing a traceability environment that is more supportive of change [6][9]. Instead of establishing direct and tightly coupled links between requirements and dependent entities, links are established through an event service. An entity such as a scenario that is dependent upon a requirement subscribes to that requirement in order to receive event notifications when the requirement changes. Changes applied to requirements result in the publication of an event message to the event server and the subsequent forwarding of that message to all dependent entities. Case-studies have demonstrated EBT's ability to handle long-term change effectively [9].

This paper focuses upon the role played by the requirements specification as a publisher of events in the EBT scheme. For EBT to work efficiently event publication must be automated. Section 2 of this paper provides a brief description of the EBT method and its underlying architecture. Section 3 defines a set of change events that describe the structural evolution of a requirements specification, and section 4 identifies the lower level actions from which each of these events are composed. Section 5 defines an algorithm for monitoring

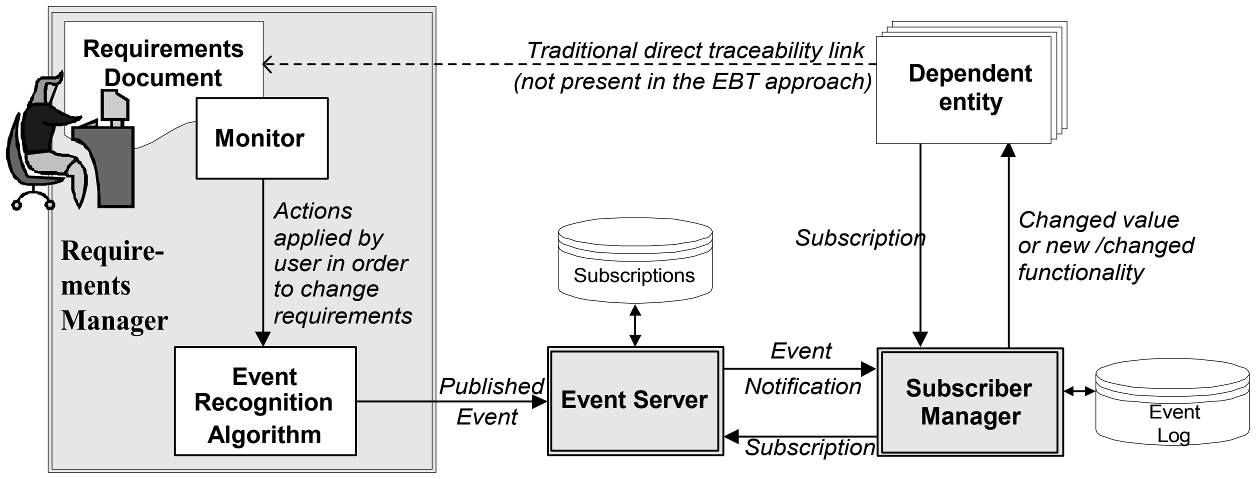
Requirements Document
Traditional direct traceability link (not present in the EBT approach)
Dependent entity
Monitor
Requirements Manager
Actions applied by user in order to change requirements
Subscriptions
Subscription
Changed value or new /changed functionality
Event Recognition Algorithm
Published Event
Event Server
Event Notification
Subscription
Subscriber Manager
Event Log

Figure 1. The role of event recognition within the event-based traceability scheme.

change actions within a requirements management tool, and identifying and outputting change events. Section 6 provides an example of applying the algorithm and section 7 concludes with a discussion of the algorithm and its effectiveness both within the EBT scheme and as a general mechanism for managing change.

2. Event Based Traceability

An EBT architecture, as depicted in Figure 1, contains three main components. The requirements manager is responsible for managing requirements, monitoring changes to those requirements, and for publishing change event messages to the event server. The event server is responsible for establishing traceability by handling initial subscriptions placed by dependent entities. It also listens for event notifications from the requirement manager(s) and forwards event messages to relevant subscribers. Finally, the subscriber manager listens on behalf of the subscribers that it manages for event notifications forwarded by the event server. Depending upon both the event and subscriber type, the manager either stores the incoming event message in an event log for later human-supported resolution, or else processes it automatically according to a set of predefined rules.

In almost any traceability scheme there are both direct and indirect dependencies upon requirements. For example, a high-level design document might link directly to a requirement, whilst a low level design document might link to the high level document. Change event notifications sent to a direct dependent must therefore be resolved by that entity and forwarded to its own dependents. A major consideration in defining a set of standard event messages is to balance the need for message simplicity with the need to minimize the number of events by packaging a meaningful amount of information into each event.

3. Change events

The problem of defining change events during requirements evolution is quite similar to that of identifying the steps needed to edit a document. During the 1970s and 80s several researchers studied a problem known as "Tree-to-tree" correction and identified a set of primitive steps that could be applied sequentially to one hierarchical object in order to produce another [10][11]. Although approaches varied, the basic steps included actions such as *insertTree, deleteTree, insertInternalNode, deleteInternal-Node, changeNode,* and *swapSubTree.* In the same way, a set of change events can be defined to describe the structural evolution of a requirement set.

Pinheiro et al [12], and Ramesh [2] both identified traceable relationships such as *derive, refine, replace, specify, test, extract, part-of*, and *abandon*. Through observing the evolution of a set of over 300 requirements, integrating the relationships documented by Ramesh and Pinheiro, and considering the actual task of updating dependent entities in response to event notifications, we defined the following non-exclusive set of change events. The symbol $\rightarrow$ denotes a change from the entity or entities on the left hand side, to those on the right hand side. Additional event types could also be considered.

1. Create a new requirement. $\rightarrow r_i$
2. Inactivate a requirement. $r_{i.Status(Active)} \rightarrow r_{i.Status(Inactive)}$
3. Modify an attribute value. $r_{i.Status(old)} \rightarrow r_{i.Status(new)}$
4. Merge two or more requirements $r_i + r_j + ... + r_m \rightarrow r_n$
5. Refine a requirement by adding an additional part. $r_i \rightarrow r_i + r_j + ... + r_n$

6. Decompose a requirement into two or more parts. $r_i \rightarrow r_j + r_k + ... + r_n$
7. Replace one requirement with another (For example: when a new prioritized requirement conflicts with an existing one) $r_i \rightarrow r_j$

4. Change actions

Most requirement management tools support actions such as adding and deleting requirements and links, attaching attributes to requirements and links, and setting the values of those attributes. In various combinations, these actions are used to apply high-level changes such as to decompose or merge requirements. The following abbreviations and terminology are used throughout the remainder of the paper to describe change actions.

- R = requirement, A = attribute, V = value, L = link
- CreateRequirement(String) → R
  creates a requirement R from a text string.
- SetRequirementAttribute(R,A,V)
  sets the attribute A on requirement R to the value V.
- CreateLink(R,R') → L
  creates a link L from requirement R to R'
- SetLinkAttribute(L,A,V)
  sets the attribute A on link L to the value V.

4.1 Change events defined as actions

In this section, each event is defined in terms of its lower level actions.

- **New event**

New creates a new requirement from a text string.

```
CreateRequirement(String) → $R_{new}$
```

- **Inactivate event**

Inactivate sets the status of a requirement to inactive.

```
SetRequirementAttribute(R, "Status", "Inactive")
```

- **Modify event**

Modify changes the value of an attribute. The attribute must be a member of a user-defined set of critical attributes. Critical attributes are defined by the user, and represent attributes that trigger an event if their value is changed. For example, performance related requirements that drive executable performance models could be stored as attributes in the requirements [6].

```
SetRequirementAttribute(R, Attribute, Value) |
Attribute ∈ { User-defined critical attributes}
```

- **Merge event**

A merge event merges two or more requirements into one new requirement.

```
CreateRequirement( String ) → $R_{new}$
for i = 1 to no. of requirements to merge
    CreateLink( $R_{new}$, $R_{old-i}$) → $L_i$
    SetLinkAttribute( $L_i$, "LinkType", "Merge")
    SetRequirementAttribute($R_{old-i}$ , "Status", "Inactive")
end for
```

- **Refine event**

A refine event takes a requirement and augments it by adding one or more additional parts.

```
for i = 1 to no. of refined parts
    CreateRequirement(String) → $R_i$
    CreateLink($R_i$, $R_{old}$) → $L_i$
    SetLinkAttribute( $L_i$, "LinkType", "Refine")
end for
```

- **Decompose event**

A *decompose* event takes one requirement and decomposes it into two or more parts. The actions are identical to those of the *refine* event except for the addition of the action that sets the original requirement as inactive.

```
for i = 1 to no. of decomposed parts
    CreateRequirement(String) → $R_i$
    CreateLink($R_i$, $R_{old}$) → $L_i$
    SetLinkAttribute($L_i$, "LinkType", "Refine")
end for
SetRequirementAttribute($R_{old}$, "Status", "Inactive")
```

- **Replace event**

A *replace* event replaces one requirement with another requirement – for example in response to a trade-off analysis in which the two requirements conflict and cannot co-exist.

```
CreateRequirement(String) → $R_{new}$
CreateLink($R_{new}$, $R_{old}$) → $Link_{new}$
SetLinkAttribute( $L_{new}$, "LinkType", "Replace")
SetRequirementAttribute($R_{old}$ , "Status", "Inactive")
```

4.2 Valid Input Actions

Because a CreateLink(R,R')→L action always precedes a SetLinkAttribute(L,A,V) action and all events dependent upon link types can be recognized purely through the SetLinkAttribute(L,A,V) actions, it is not necessary to monitor CreateLink(R,R') → L actions. In contrast, the CreateRequirement(String) → R action must be monitored because the *new* event is dependent only upon that action. The three types of monitored input actions are therefore:

- CreateRequirement(String) → R
- SetRequirementAttribute(R,A,V) | (A = "Status" $\wedge$ V $\in$ { "Active", "InActive"}) $\vee$ A $\in$ {User-defined set of performance-related attributes)
- SetLinkAttribute(CreateLink(R,R'),A,V) | A = "LinkType" $\wedge$ V$\notin$ { "Merge", "Refine", "Replace"}

Another important observation is that the defined events can be represented as a hierarchy of composite and primitive events. For example the *decompose* event is constructed from a *refine* event plus an *inactivate* event, whilst the refine event is composed from a series of *new* events. In order to avoid incorrect recognition of events that are still being processed, event recognition should therefore occur over a user-defined session that only ends when all events that have been started have also been completed.

4.3 Constraints

The following constraints upon input actions and output events are expressed using temporal logic in which the symbol ω represents the "weak until" operator.

Basic input constraints – enforced automatically by many requirements management tools.

- $\forall R$ [! SetRequirementAttribute(R, A, V) ω CreateRequirement(String) $\rightarrow$ R]
- $\forall R$[! CreateLink(Ri,Rj) ω (CreateRequirement(String) $\rightarrow$Ri) $\wedge$ CreateRequirement(String) $\rightarrow$ Rj)]
- $\forall L$ [! SetLinkAttribute(L,A,V)ω CreateLink(R,R')$\rightarrow$L]

Cyclic dependencies - Cyclic dependencies are not valid inputs as they result in non-deterministic ordering of outputs.

- $\forall R$ [! (SetLinkAttribute(Link[Ri,Rj],A,V) $\wedge$ SetLinkAttribute(Link[Rj,Rk],A,V) $\wedge$ SetLinkAttribute(Link[Rk,Ri],A,V))] | (A = "LinkType") $\wedge$ (V $\in$ {"Merge"| "Refine"|"Decompose" | "Replace"})

Data dependencies - Output events must be ordered according to data dependencies between events. If E = {Ei, Ej, ...En} is the set of output events, then an event Ej may not have a requirement R on its right hand side (RHS) if another event Ei has previously used the same requirement on its left hand side (LHS) unless Ej is a refine or modify event and R is also on the LHS of Ej.

- $\forall R$ $\forall E$ [! Ei ω Ej] | (R$\in$Ej.RHS)$\wedge$(R $\in$Ei.LHS) $\wedge$! (R$\in$Ej.LHS)

5. Change event recognition algorithm

The change event recognition algorithm is described below using pseudocode. The event list is used to process the actions and events. Constraints are described in section 4.3 and are therefore not restated here.

The main event recognition algorithm monitors the input actions. CreateRequirement(String)$\rightarrow$R actions are immediately added to the event list. SetLinkAttribute(L,A,V) actions are handed to the *LinkHandler* and SetRequirementAttribute(R,A,V) actions are handed to the *RequirementHandler*. All other input actions are ignored. The main algorithm is shown below and the *LinkHandler* and *RequirementHandler* algorithms are described in the following sections.

Inputs: Change Actions
Outputs: Change Events

Event Recognition

```
while sessionIsActive do
    Action = GetNextChangeAction()
    Case Action
    Case Action == "CreateRequirement(String)→ R"
        Add New event to the event list
    Case Action == "SetLinkAttribute (L, A, V)" then
        Pass SetLinkAttribute(L,A,V) to Link Handler
    Case Action =="SetRequirementAttribute(R,A,V)"
        Pass SetRequirementAttribute(R,A,V) action to
        the Requirement Handler
      Default
        // Do Nothing
    end case
end while
output change events
```

The link handler is responsible for creating and changing relationships between requirements. This algorithm does not describe how corrections would be handled if the user had previously established an incorrect link. It is assumed that this could be supported through an undo feature. As a link can indicate that previously identified *new* or *inactivate* events are now to be incorporated into a higher level event, the link handler must first search for related *new* or *inactivate* events and delete them. If the link type is "refine", and a related *inactivate* event is found, the event type is promoted to "decompose." The link handler searches for an existing event. If found, the newly linked requirement is added as an additional component to that event, otherwise an event of the type described in the link is added to the event list.

Link Handler

```
Search event list for related new event
Delete the new event

Search for a related inactivate event
if inactivate event is found
    Delete the inactivate event
    if LinkType = "Refine"  then
        Promote the LinkType to "Decompose"
    end if
end if

Search for an event to which this action might belong
if event is found
    Add new action to the existing event
else
    Locate the correct position to insert the event
    Insert the event into the event list
end if
```

The requirements handler receives actions either indicating that the requirement has been inactivated, or that one of the user-defined critical attributes has been modified. An inactivate action results in one of three possibilities. It can promote an existing *refine* event to a *decompose* event, can cause a primitive *inactivate* event to be added to the list, or may be ignored if the requirement has already been linked to a higher level event.

Requirements Handler

```
if status attribute is set to "inactive"
    Search for a related refine event
    if refine event is found
        Promote the refine event to a decompose event
    else
        Search for a related merge event
        if merge event is found
            Do nothing – because the inactivate is part of
            the previously recognized merge event
        else
            Search for related replace event
            if replace event is found
                Do nothing – as the inactivate is part of
                the previously recognized replace event
            else
                Add an inactivate event to the event list
            endif
        endif
    endif
else if Attribute ∉ User-defined set of critical attributes
    Add a modify event to the event list.
end if
```

	Change Event	Event Type
1.	R5 → R11 + R12	Decompose
2.	R12 + R2 → R16	Merge

Table 1. Change events.

Event # 1. R5 → R11 + R12	
CreateRequirement(R11)	1
CreateRequirement(R12)	2
SetLinkAttribute(Link(R11,R5),Linktype,"Refine")	3
SetLinkAttribute(Link(R12,R5),Linktype,"Refine")	4
SetRequirementAttribute(R5,Status, "Inactive")	5
Event # 2. R12 + R2 → R16	
Requirement.Create(R16)	6
SetLinkAttribute(Link(R16,R12),Linktype,"Merge"))	7
SetLinkAttribute(Link(R16,R2),Linktype,"Merge"))	8
SetRequirementAttribute(R12,Status, "Inactive")	9
SetRequirementAttribute(R2,Status, "Inactive")	10

Table 2. Change actions for events 1 and 2.

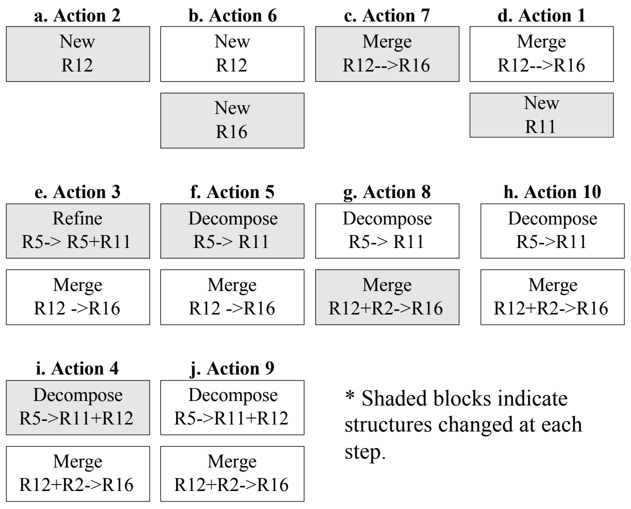

Figure 2. An example of event recognition.

6. An example

The following example, uses the change trace listed in Table 1 to demonstrate the ability of the algorithm to translate a set of input actions into change events. The actions for each of the two events are depicted in Table 2. Each action is assigned a unique action number in the rightmost column. In this example, both events are composed of 5 actions. Figure 2 depicts the event list data structure following each action as it occurs. To illustrate the algorithm's ability to handle interwoven actions, the actions are applied in the sequence order 2, 6, 7, 1, 3, 5, 8,10, 4, 9. This is one of many feasible input sequences that would meet the defined constraints.

Executing the algorithm, actions 2 and 6 add the two events New(R12) and New(R16). Action 7 then establishes a merge link between R12 and R16, and so the related *new* events are removed from the event list, and the *merge* event added. Action 1 then adds *new* R11. Action 3 establishes a refine relationship from R11 to R5, and in order to comply with the data dependency constraint on the ordering of output events, the Refine R5 → R5 + R11 is inserted into the event list before the existing merge. The related *new* R11 is removed from the list. Action 5 sets requirement R5 to inactivated, thereby triggering the promotion of the refine event to a decompose event. Action 8 adds an additional component to the existing merge event. Action 10 has no effect because R2 was already recognized as part of a merge event. Action 4 adds an additional component to the decompose event, and finally action 9 has no effect because again R12 was already recognized as part of the merge event. The result is the correct recognition of the two events.

When the user ends the session the change events are output. An example of the event message published for change event (R5→R11 + R12) is: "Decompose | R5, MNet, \Functional\Whiteboard, The user shall be able to draw shapes on the whiteboard | R11, MNet, \Functional\Whiteboard, The user shall be able to draw free hand shapes on the whiteboard using a drawing tool | R12, MNet, \Functional\Whiteboard, The user shall be able to select pre-drawn shapes from a shape pallete to place on the whiteboard | Jane.Huang | Tue Sep 25 12:10:09 PDT 2001" Additional information concerning the link type can also be attached to each message.

EBT has also been utilized in more extensive case studies involving the introduction of twenty non-trivial change proposals [9]. These proposals translated into 103 change actions and resulted in the recognition of 28 change events and the propagation of 187 event messages throughout the system. Without the use of standard event messages, a staggering 821 messages would have been triggered. The event messages were then used to support the process of updating the dependent entities to reflect the changes. This was accomplished without undue difficulty, demonstrating that at a semantic and logical level the event messages carried sufficient information to accomplish their tasks. Additional experiments were conducted in which the *modify* event was used to support speculative queries. Quantitative performance related requirements were linked through the EBT scheme to relevant performance models. When values in requirements were changed, event messages were published and forwarded by the event server to the dependent performance models, which were then automatically re-executed by the subscriber manager using the changed data values [6]. Results from all re-executed models were then reported.

The event recognition algorithm can be implemented in any environment in which it is possible to monitor change actions and to make system calls. For example DOORS supports this type of algorithm through its DXL scripted triggers. These DXL scripts can monitor changes and are capable of making system calls to executable files.

7. Conclusions

The event recognition algorithm described in this paper is fundamental to the effectiveness of the EBT approach. Without the ability to automatically recognize change events as they occur, the requirements engineer would be burdened with the responsibility of manually triggering event notifications, introducing the likelihood of errors and omissions. In the existing EBT prototype, event publication was simulated through a GUI that interfaces with requirements in DOORS, however future versions of the prototype will implement the algorithm described in this paper.

Even without the EBT traceability environment, change event recognition is useful because it can provide a high-level documentation of the changes implemented in the requirements specification. This complements existing change management practices by supporting the process of updating entities related to the change. Future work will focus on implementing the algorithm in a broader range of requirements management environments, extending it to capture implicit traceability links such as those established through the positioning of requirements within a hierarchical scheme, and developing related visualization tools.

References

[1] O. Gotel, and A. Finkelstein, "An Analysis of the Requirements Traceability Problem", *Proc. of the 1st Int' l Conf. on Requirements Engineering,* 1994, pp. 94-101.
[2] B. Ramesh, and M. Jarke, "Toward Reference Models for Requirements Traceability", *IEEE Trans. on Software Eng.,* Vol. 27, No. 1, Jan. 2001, pp. 58-92.
[3] The Standish Group, *Chaos Report*, 1995. Available online at: www.standishgroup.com/visitor/chaos.htm
[4] K. Pohl, *Process-Centered Requirements Eng.,* Advanced Software Development Series, Research Studies Press Ltd. Wiley & Sons, Herts, UK, 1996.
[5] B. Ramesh, C. Stubbs, T. Powers, and M. Edwards, "Implementing Requirements Traceability: A Case Study", *Annals of Soft. Eng.,* Vol. 3, 1997, pp. 397-415.
[6] J. Cleland-Huang, C. K.Chang, H. Hu, K. Javvaji, G. Sethi, and J. Xia, "Requirements Driven Impact Analysis of System Performance", *IEEE Proc. of the Joint Conf. on Requirements Eng.*, Essen, Germany, Sept 2002.
[7] SE Tools Taxonomy – Requirements Traceability Tools, *Int' l Council on Systems Engineering,*Avail. online at http://www.incose.org/tools/tooltax/reqtrace_tools.html
[8] B.Azelborn, "Building a Better Traceability Matrix with DOORS", *Telelogic INDOORS US 2000*. Available online at: http://www2.telelogic.com/doors/index.cfm
[9] J.Cleland-Huang, *Robust Requirements Traceability for Handling Evolutionary and Speculative Change*, PhD dissertation, University of Illinois at Chicago, 2002.
[10] D. Barnard, G. Clarke, and N. Duncan, "Tree-to-tree Correction for Document Trees", *Technical Report 95-372*, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada. Available online at: http://www.cs.queensu.ca/ TechReports/ Reports/1995-372.pdf.
[11] S. Selkow, "The tree-to-tree editing problem", *Information Processing Letters,* Vol. 6, No. 6, Dec. 1977, pp. 184-186.
[12] F.A.C. Pinheiro and J.A.Goguen, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, Vol. 13, No. 2, Mar. 1996, pp. 52-64.

Acknowledgment

This research was partially funded by NSF grant CCR-0098346.