Event-Based Traceability for Managing Evolutionary Change

Jane Cleland-Huang, *Member*, *IEEE Computer Society*,
Carl K. Chang, *Fellow*, *IEEE*, and Mark Christensen, *Member*, *IEEE*

Abstract—Although the benefits of requirements traceability are widely recognized, the actual practice of maintaining a traceability scheme is not always entirely successful. The traceability infrastructure underlying a software system tends to erode over its lifetime, as time-pressured practitioners fail to consistently maintain links and update impacted artifacts each time a change occurs, even with the support of automated systems. This paper proposes a new method of traceability based upon event-notification and is applicable even in a heterogeneous and globally distributed development environment. Traceable artifacts are no longer tightly coupled but are linked through an event service, which creates an environment in which change is handled more efficiently, and artifacts and their related links are maintained in a restorable state. The method also supports enhanced project management for the process of updating and maintaining the system artifacts.

Index Terms—Change management, traceability, requirements management, evolutionary change, software maintenance, impact analysis.

1 Introduction

In recent years, the role of requirements engineering has taken a more central position in the overall software lifecycle. Software engineers have developed a deeper understanding for the importance of eliciting and documenting a complete and accurate set of requirements to act as a driving force throughout the modeling, development, testing, and maintenance of software systems. By creating traceability links between requirements and other artifacts, a number of software engineering activities can be supported. These include requirements validation, scenario based test-case generation, recording rationales, impact analysis of change, and guarding against gold-plating [1], [2], [3].

One of the greatest challenges of maintaining a traceability scheme is the fact that the artifacts being traced continue to change and evolve as the system is developed [4], [5], [6]. Studies have shown that change can account for 40 to 90 percent of total development costs [7], [8], [9], [10]. Related difficulties have been observed for many years, for example, Martin and McClure [11] discussed the difficulties involved in maintaining external software documentation, and these same problems continue to be experienced by practitioners today.

Software configuration management (SCM) techniques are commonly used to control the evolution of software systems through providing version control, configuration identification, and support for accounting activities related to change management [12], [13]. The strength of SCM is its ability to track the configuration of multiple versions of a single product and to control changes to the product. Changes within individual objects are documented using evolution graphs [13] while traceability techniques are used to identify all artifacts that should be updated when a change is introduced. Unfortunately, there is a tendency in even the best traceability schemes for links to fail to keep pace with the evolving system, resulting in the gradual erosion of the traceability infrastructure and its eventual failure to reliably represent the current state of relationships [14], [15], [16].

In 1994, Gotel and Finkelstein conducted an extensive survey of traceability problems, in which she identified several contributing factors. These included use of informal development methods and failure to follow standard practices; insufficient resources, time, and support allocated to traceability; lack of clarity concerning roles played by individuals in the traceability process, and lack of ongoing cooperation and coordination between people responsible for various traceable artifacts; an imbalance between the amount of work involved in establishing and maintaining traceability and the perceived benefits; difficulty in obtaining necessary information in order to support the traceability process; and, finally, lack of training in traceability practices [3], [17], [18]. These problems have been exacerbated by the challenges of today's globally distributed development environment. In many projects, a dichotomy exists between what should take place and what actually does take place. This, together with the fact that the "traceability problem" has been around for such a long time without being solved, is an indication of the difficulties involved.

- *J. Cleland-Huang is with the School of Computer Science, Telecommunications, and Information Systems, DePaul University, 243 S. Wabash Ave. Chicago IL 60604. E-mail: jhuang@cs.depaul.edu.*
- *C.K. Chang is with the Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50011. E-mail: chang@cs.iastate.edu.*
- *M. Christensen is available at 36W500 Wild Rose Rd. St. Charles, IL 60174-1149. E-mail: markchri@concentric.net.*

Manuscript received 7 Dec. 2001; revised 4 Mar. 2003; accepted 15 Apr. 2003. Recommended for acceptance by D. Mandrioli.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 115529.

Several researchers have found that inadequate traceability continues to be a major contributing factor in project over-runs and failures [19], [20]. This paper takes a more in-depth look at problems related to forward traceability and proposes a new traceability method based upon event notification to alleviate some of these problems. In event-based traceability (EBT), requirements and other software engineering artifacts are linked through publish-subscribe relationships in which requirements and other change instigators take on the role of publishers, while dependent artifacts act as subscribers. When requirements are changed, events are published to an event server and related notifications sent to all dependent subscribers. Messages carry sufficient information to provide meaningful semantics about each event in order to support the update process.

To understand exactly how EBT alleviates the maintenance problem, the following section discusses existing traceability methods, examines the ways in which traceability maintenance activities typically fail, and identifies the areas in which EBT can help. Following that, the remainder of this paper describes the EBT approach in more detail. Sections 3 and 4 introduce and discuss the proposed event-based approach to traceability, and Section 5 describes a prototype tool that we developed and describes an example of using EBT to manage change events. Section 6 then concludes with an evaluation of Event-Based traceability and a discussion of future work.

2 Traceability Methods

Traceability links define the relationships that exist between the various artifacts of the software engineering process [2], [21]. Artifacts include objects such as requirements, code modules, designs, test cases and results, and various other entities that represent characteristics and behavior of the system. Artifacts are also referred to as software configuration items [13]. The following paragraphs more formally define an artifact and a link.

Definition 1. *An **artifact** is a piece of information produced or modified as part of the software engineering process [22]. Artifacts take a variety of forms including models, documents, source code, test cases, and executables. Such artifacts, in whole or in part, form the traceable objects of the system. Let* $A = \{a_1, a_2, a_3, \ldots, a_n\}$ *be the set of all identified artifacts in the system. Let* $R = \{r_1, r_2, r_3, \ldots, r_n\} \subset A$ *be the subset of artifacts that are requirements.*

Definition 2. *A **link*** $Link(a, a')$ *represents an explicit relationship defined between two artifacts* a *and* a'*. If* a *and* a' *are directly linked as in* $a \rightarrow a'$*, where* $\rightarrow$ *indicates a link, then the link is said to be **direct**, and if* a' *and* a'' *are indirectly linked through one or more intermediate artifacts, as in* $a \rightarrow a' \rightarrow a''$*, then the link is said to be **indirect**. Let* $\mathrm{a} => \mathrm{a}''$ *represent an indirect link from* a *to* a''*. An artifact (such as* a'*) through which an indirect link is established is said to be an **intermediate** artifact. The direction of the link indicates that the link is established from the artifact on the left-hand side (LHS) to the one on the right-hand side (RHS), such that the LHS artifact exhibits a **dependency** upon the RHS artifact.*

A link may be annotated with one or more attributes that describe some characteristic of the link [23], [20]. Many different techniques are used to represent traceability links including standard approaches such as matrices [21], [19], hypertext links [24], [25], graph-based approaches [26], formal methods [27], [28], and a variety of dynamic schemes [15], [29]. Automated support for traceability includes general-purpose tools such as word processors, spreadsheets, or database systems; as well as special purpose traceability or requirements management tools such as DOORS™ [30], [31], Requisite Pro™ [32], [33], CRADLE™, and SLATE™ [34]. These tools typically support traceability through implementing approaches such as matrices or hypertext links and differ in the quantity and diversity of traceable information they are able to support.

The nondynamic methods establish traceability through the use of direct links between artifacts. However, this means that as the system grows in size and complexity, and the volume of artifacts increases, the number of interdependencies and related links can grow exponentially. It is well-known that any type of system that is tightly coupled in this way can become extremely brittle and hard to change [35], [36], [37], [38], so it is hardly surprising that these tightly linked traceability infrastructures do not fare well under the inevitable changes that occur during the lifetime of a software system. A typical consensus among practicing software engineers was summarized by Brian Azelborn, a 15 year veteran of Rockwell Collins, who stated that "it is often the case that relationships are not created between requirements" and that "existing relationships are not maintained" [30]. In addition to Gotel's correct observation that the cause of forward traceability problems can be primarily attributed to lack of adherence to standard procedures, we can also state that even when strict processes are followed, the task of maintaining a tightly coupled set of traceability links by use of matrices, hyperlinks, or graphs remains a very difficult one.

Reexamining Gotel's findings concerning the difficulty of maintaining traceability links, it is apparent that several of them are related to the problem of tight coupling. In addition, Gotel identified the lack of ongoing cooperation and coordination between people responsible for various traceable artifacts as one of the problems. The coordination and coupling factors compound one another, increasing the difficulty of maintaining the artifacts in a consistent state. Thus, in a traditional format such as a traceability matrix or graph, links must be carefully maintained through coordinating the updating of all impacted artifacts. In contrast, EBT relaxes the coordination efforts needed to maintain artifacts and resolve links in response to a change, reducing the compounding effects.

2.1 Handling Change

To understand why traceability maintenance is so difficult, it is necessary to first examine the ways in which the traceability infrastructure is affected by change.

Definition 3. *A **change C** introduced to an artifact a can be in one of two phases. A **proposed** change implies that impact analysis should be performed to determine how change C would impact the existing system, whereas an **implemented** change implies that all impacted artifacts and their related links should be updated to reflect the change. A proposed change does not necessarily result in an implemented change.*

When a change C is introduced to an artifact a, the remaining artifacts of the system can be categorized in the following ways according to their relationship to a.

Definition 4: Linked artifact. *Let $Linked(a) \subseteq A$ be the set of all artifacts linked either directly or indirectly to artifact a through one or more explicitly defined traceability links.*

Definition 5: Related artifacts. *Let $Related(a) \subseteq A$ be the set of artifacts intrinsically related to artifact a either directly or indirectly, whether or not a traceability link has been established between them. There are two primary reasons that $Linked(a) \neq Related(a)$. First, it is not desirable to represent every conceivable artifact relationship with a link, as this would lead to an unwieldy tangle of links, many of which would be unlikely to ever be used [18], [20], [23]. Decisions about which related artifacts to link should be based upon well-defined project level strategies. Therefore, let $StrategyRelated(a) \subseteq Related(a)$ represent the set of artifacts that according to project level strategies should be explicitly linked to artifact a. Ideally, $Linked(a) = StrategyRelated(a)$. Second, certain links that should exist may be absent due to earlier failure to update and maintain the traceability infrastructure. For this reason, it is often the case that $Linked(a) \subset StrategyRelated(a)$.*

Definition 6: Impacted Artifacts. *Let $Impacted(C, a) \subseteq Related(a)$ be the set of artifacts impacted by a proposed change C upon artifact a.*

Definition 7: Identified Artifacts. *Let $Identified(C, a) \subseteq A$ be the set of artifacts identified as impacted by a proposed change C upon artifact a. Unless the developer manually identifies artifacts external to the traceability scheme, $Identified(C, a)$ should be a subset of $Linked(a)$.*

Definition 8: Updated Artifacts. *Let $Updated(C, a, t) \subseteq Identified(C, a)$ be the set of artifacts updated as a result of implementing change C upon artifact a and following a point in time t at which all artifacts and links related to change C are updated.*

Furthermore, certain artifacts play a more critical role in system wide impact analysis than others. They are more centrally situated within the physical traceability graph or tree and when inadequately maintained may inhibit the ability to identify the full complement of artifacts during future impact analysis activities. These artifacts are defined in terms of their position within a series of traceability paths.

Definition 9. *A **traceability path** TP is an ordered set of links, in which indirectly linked artifacts a and a_{j+n} are connected through a series of directly linked intermediate artifacts, such that $TP(a_j, a_{j+n}) = \{Link(a_j, a_{j+1}), Link(a_{j+1}, a_{j+2}), \ldots, (a_{j+n-1}, a_{j+n})\}$ represents the traceability path from artifact a_j to a_{j+n}. A **requirements traceability path** represents the special case for which a_j is a requirement. The length of the shortest traceability path from an artifact to its closest requirement specifies the **level** of that artifact. An artifact directly linked to a requirement is said to be at level 1, while an artifact with one intermediate artifact is said to be at level 2, and so on. An artifact is said to be **critical** if it resides as an intermediate artifact on multiple requirements traceability paths because failure to maintain it would result in failure to maintain other artifacts at lower levels on the same traceability path.*

Many different scenarios occur during the change maintenance process, some of which result in successful outcomes and others in failure. Four scenarios are discussed below in order to categorize the types of traceability failure that might occur for a given change C introduced to artifact a.

Scenario 1: No Errors. This scenario is represented by the following relations between sets of artifacts, namely, that all impacted entities are identified and updated, and non-impacted or nonrelated entities are neither identified nor updated.

- $Impacted(C, a) = Identified(C, a) = Updated(C, a, t)$ when change C is fully implemented.
- $Impacted(C, a) = Identified(C, a)$, and $Updated(C, a, t) = \emptyset$ prior to the beginning of the implementation of change C or if a decision is made to not implement C.
- $Linked(a) = StrategyRelated(a)$ (No unintentionally missing links).

In the first case, updating of impacted artifacts is essential to the future integrity of the traceability scheme.

Scenario 2: Identification Errors. Identification errors occur when artifacts that should be included in the impact analysis are not identified. Such errors are normally a result of broken or missing traceability links, or of misinterpretation of the content of one of the two artifacts connected by the link.

- $Impacted(C, a) \supset Identified(C, a)$. These errors imply that impact analysis will be incomplete, and that a change may be introduced into the system without fully understanding its implications and side effects. Identification errors increase the likelihood of future errors, because $\neg Identified(C, a)$ automatically implies $\neg Updated(C, a, t)$, which, in turn, means that artifacts impacted by a future overlapping change C' related to artifacts in $Identified(C, a) \cap Updated(C, a, t)$ have a reduced chance of being identified.

Scenario 3: Update Errors. Update errors occur when impacted artifacts and their related links are not updated to reflect an implemented change. However, no error occurs if change C is not actually implemented.

- $Impacted(C, a) \supset Updated(C, a, t)$ when change C is fully implemented.

Scenario 4: Inclusion Errors. Errors of inclusion occur when an artifact is identified as being impacted even though it is not.

- $Identified(C, a) \cap \neg Impacted(C, a) \neq \emptyset$.

Because this class of error occurs more rarely and does not create significant traceability problems, it is not considered any further in this paper.

Definition 10. *An artifact is said to be in a **consistent** state when its state and the state of its related links accurately represent the current state of the system configuration. When an artifact is not in a consistent state, it is said to be in an **inconsistent** state. For example, an artifact is in an inconsistent state when $Impacted(C, a) \supset Updated(C, a, t)$ for an implemented change C.*

2.2 EBT Support for Change

Most traceability tools and methods successfully support the identification of impacted artifacts when an adequate and accurate set of traceability links exists. However, most methods and tools do not provide any type of support for ensuring that artifacts and related links impacted by a change are updated in a timely fashion. As a result, update errors occur which adversely affect the ability to identify artifacts impacted by future changes and, therefore, lead to identification errors in the future.

One reason that developers may fail to update artifacts relates to the way many projects are managed. Artifacts are often made available for examination and analysis by all team members, while authorization to update artifacts is limited to specific workers [22]. In this type of environment, a team member may perform impact analysis on a certain artifact, but then fail to ensure that the artifact is updated following the implemented change. This particular problem relates to Gotel and Finkelstein's observations [3] of insufficient coordination and communication between team members, limited perception of the benefits of maintaining traceability links, and lack of resources needed to complete the task. As a result, the artifact and its related links are not updated to reflect the change, causing major problems if the artifact resides on a critical path.

EBT directly alleviates the problems caused by poor communication and coordination because an event server automates the role of change notification. "Owners" of specific artifacts can update them at their convenience by examining the change event logs. To counteract the danger that the entire system might slip into an obsolete state, EBT supports a heightened level of project management, in which areas that have not been maintained are clearly visible. EBT also addresses the problem in which developers fail to perceive the benefits of investing time and effort into maintenance activities because in EBT it is possible to defer update of noncritical artifacts until the time at which they are actually needed and the benefits are obvious to all [39].

3 Event-Based Traceability

To alleviate the **update** problem discussed in the previous sections, we propose a traceability scheme based upon Event Notification using the related Event Notifier design pattern [35], [36]. This pattern addresses problems related to handling change and provides a mechanism for minimizing "the number of dependencies and interconnections between objects to keep the system from becoming brittle and hard to change [36]." The major advantage of implementing an event-based approach to support traceability is that the event server handles the coordination between "owners" of various artifacts and supports system wide visualization and management of the update state of all artifacts.

Other researchers have proposed the use of workflow or process-based approaches to software engineering. Workflow management systems (WFMS) define workflows and their implementation functionality in terms of scheduling, execution, and control. In an event-based WFMS, all interesting activities are expressed as events, and operations are defined as reactions to event occurrences. The EvE (EVent Engine) project [40], [41] is a concrete example of an implemented WFMS. The EvE event engine is notified when a primitive event occurs such as a request to execute or terminate an activity. Once it determines that a composite event has occurred, it then selects a processing entity to handle the event and notifies the selected processing entity by updating its work list. In addition to EvE, WFMSs have been implemented in a number of other projects such as TRAM [42], SEAMAN [43], and SWORDIES [44], and in fact the TRAM engine is currently implemented within SLATE (System Level Automation Tool for Engineers) as an event engine situated between requirements and design artifacts. However, to our knowledge no one has previously considered the use of an event engine to solve the traceability update problem.

The KBSA (knowledge-based software assistant) [45] and related development environments such as ARIES [46], [47], conceptDemo [48], [49], and Gist [50] support a concept known as "evolution transformations" that are able to carry out stereotypical modifications on a requirements specification and its related artifacts. In a KBSA environment, the analyst specifies the desired effect of a change and then is guided through the process of selecting an appropriate transformation. However, these transformations are primarily applied to formally specified requirements, whereas EBT is applicable within a much broader range of software engineering environments, including less formal ones.

EBT not only implements an event engine, but also provides project management support in the form of a distributed project-wide process guidance system. Gotel identified lack of training and failure to follow standard procedures as two of the root causes of the traceability problem. Several researchers have shown that process guidance implemented within a process-centered environment (PCE) can alleviate both of these problems [51], [52]. PCEs are composed of a modeling domain in which processes are defined and modeled; an enactment domain, which interprets the project instantiated model in order to guide, control, and monitor activities; and a performance domain in which these activities are actually implemented [53], [54]. Project management support within a PCE is primarily concerned with the coordination of people's tasks, ensuring that they are performed in a timely manner

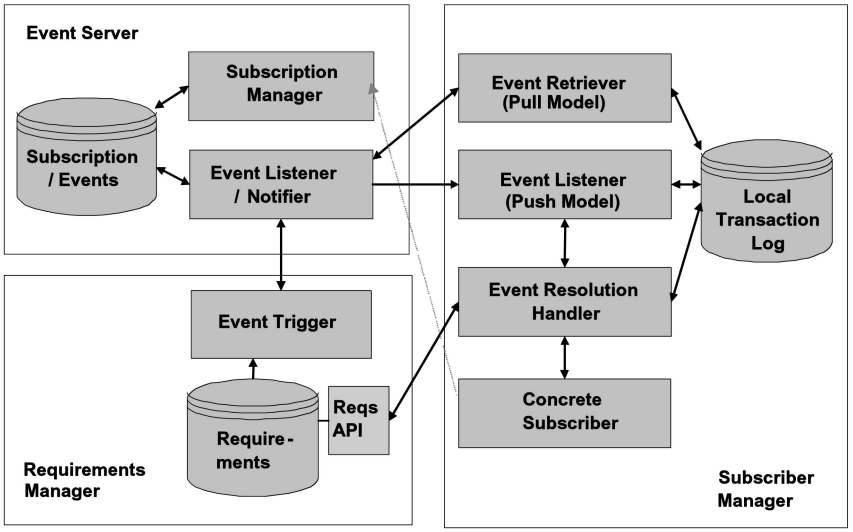

Fig. 1. System level model of Event-Based Traceability.

and in a logical sequence, rather than ensuring that the tasks necessary to perform the project's processes are correctly identified. The use of a PCE reduces the training needs of developers while assisting them to successfully follow predefined procedures.

Pohl et al. proposed the need for a closer integration of performance and enactment domains within a Process Integrated Environments (PIE) [52]. Activities are broken down into tasks and represented as method fragments, which are then allocated by the enactment domain according to the current state of the process model and the abilities of the tools in the performance domain. A PIE supports automated, guided, and enactment services. Automated services are executed without user intervention in response to a request from the enactment domain; guided services provide the user with options to select from, while enactment services enable tools to request the execution of method fragments [55]. These services therefore enable true peer-to-peer connectivity between domains and also require that the enactment domain not only issues tasks, but also monitors and controls the execution of those tasks.

EBT implements several aspects of a PIE, but distributes much of the enactment domain logic to the subscriber managers themselves, thus reducing the knowledge required by the enactment domain to understand the capabilities of each tool. Traceability links and current process policies that drive the appropriate selection of task methods for a given event are stored in tables in the event server, but, in EBT, the control for allocating specific tasks and method fragments to events is driven by enactment domains distributed among the artifacts. The activities of developers are coordinated through the use of a strongly typed messaging system, event logs, and integrated project management tools.

As depicted in Fig. 1, EBT's architecture is composed of a requirements manager, event server, and subscriber manager connected using a standard communication mechanism. The requirements manager handles the requirements and is responsible for triggering change events as they occur. The subscriber manager places initial subscriptions on behalf of the artifacts it manages, interacts with the process modeling domain to register for appropriate tasks, handles event notifications on behalf of the artifacts under its control, and supports the process of restoring an artifact and related traceability links to a current state. The event server manages subscriptions, receives event messages, customizes event notifications according to the process model and subscriptions, and forwards task directives in the form of event notification messages to the subscribers. The following sections describe each of these components and related issues in more detail.

3.1 Requirements Manager

EBT is based upon the premise that requirements evolution can be represented as a series of change events, and that an event message is published each time such a change event occurs. This concept is discussed more fully in [56]. In the 1970s, several researchers examined the similar problem of tree-to-tree correction of text documents. In this work, they identified a set of primitive steps that should be taken to convert one string or document into another [57], [58]. For example, the primitive steps of *add character*, *delete character*, and *insert character* compose the basic steps needed to convert a string, while primitive steps such as *insertTree, deleteTree, insertInternalNode, deleteInternal-Node, changeNode,* and *swapSubTree* can be used to transform an entire document. Different researchers proposed slightly different sets of primitive steps, indicating that alternate options provide viable solutions to the problem. Similarly, a requirements specification evolves through a series of primitive steps or events.

By considering interrequirement relationships [23], [36] and through observing the evolution of requirements in several projects, we identified a set of change primitives. Selection of this particular set of event types primarily took

into consideration the ultimate purpose of EBT, which is to notify dependent artifacts and developers of changes that have occurred in order to support the developer in the task of updating an artifact [56]. In the following definitions, the symbol $\rightarrow$ indicates that the requirement(s) on the left-hand side are transformed to those on the right-hand side by the change event. Each link can have a number of attributes associated with it but must always possess an attribute that indicates if the link is active or inactive.

1. **Create** a new requirement. $\rightarrow r_i$.
2. **Inactivate** a requirement.

$$r_{i.Status(Active)} \rightarrow r_{i.Status(Inactive)}.$$

3. **Modify** the value of an attribute attached to a requirement $r_{i.Status(old)} \rightarrow r_{i.Status(new)}$. (Inactivate is a special case of Modify, however, in terms of event resolution it is handled very differently, and so we prefer to distinguish between these two event types.)
4. **Merge** two or more requirements

$$r_i + r_j + \ldots + r_m \rightarrow r_n.$$

5. **Refine** a requirement by adding an additional part. $r_i \rightarrow r_i + r_j + \ldots + r_n$.
6. **Decompose** a requirement into two or more parts. $r_i \rightarrow r_j + r_k + \ldots + r_n$.
7. **Replace** one requirement with another (for example, when a new prioritized requirement conflicts with an existing one) $r_i \rightarrow r_j$.

To implement EBT effectively, it is necessary to support the automated recognition of these events as they occur in the requirements specification. Without automated recognition, the developer would be burdened with the task of identifying and publishing events, which would inevitably lead to both errors of omission and publication of incorrect events. Most requirements management tools allow the users to apply only primitive actions such as add a requirement, delete a requirement, modify requirement text, add and remove links, and add, remove, or edit an attribute attached to a requirement or to a link [34].

By describing each of the change events listed above in terms of its lower-level actions, it is possible to identify a recognizable "signature" for each event. For example, the replace event is composed of the actions:

```
create requirement R_new
create link L_new from requirement R_new to R_old
add a "linktype" attribute to L_new and define it
 as "Replace"
inactivate requirement R_old
```

while the refine event is composed of the actions:

```
for i = 1 to no. of refined parts
  create requirement R_i
  create link L_i from requirement R_i to R_old
  add a "link type" attribute to L_i and define it
   as "Refine"
end for
```

The complete algorithm for supporting event recognition is described in [56]. To support EBT, the requirements management tool must provide support for triggering events. Currently, only a few tools such as DOORS™ and SLATE™ offer this feature. DOORS™ supports triggers through the use of its C-like DXL scripts that can be triggered in response to certain changes and can make system calls to an external file that monitors the user's change actions and identifies events as they occur [56]. SLATE utilizes a full event-engine to connect requirements to a wide variety of system models. Its event-engine could be used in place of the EBT event server; however, an event recognition tool would still need to be integrated into the system to monitor events as they occur. Other tools such as RequisitePro and AnalystPro [34] keep their own change history files that could provide the session data needed by the EBT tool. Such tools that also provide an adequate API for querying historical data can also support a batch-type version of EBT event recognition.

3.2 Event Server

The event server is primarily responsible for handling subscriptions, receiving change notifications, and forwarding customized event messages to the subscriber managers of dependent artifacts. Customized messages are needed to guide update activities during event resolution. The event server stores process knowledge in terms of appropriate actions to be taken by specific subscriber types in response to standard events. At the time a subscription is placed, the centralized enactment component within the event server guides the subscriber in registering for specific task directives. For example, the process model might define suitable tasks for "test case" subscribers such as "Update artifact," "Resolve Link," "Execute test," and "Report results." The suggested tasks are dependent upon both the artifact type and the event type. A previous study showed that performance related impact analysis could be supported through speculative events that triggered tasks for injecting speculative values into performance models, reexecuting models, analyzing results, and restoring the state of the original models [59], [60], [61]. At the time the subscription is placed, the subscriber manager either accepts or rejects prescribed tasks on behalf of the subscribed artifact. The event server is discussed in more depth in Section 4.

3.3 Event Messages

When a change event occurs, the requirements manager publishes a generic event message to the event server. The message contains structural and semantic information about the change itself, including the event type {Create | Inactivate | Modify | Merge | Refine | Decompose | Replace}, requirement IDs and descriptions, and additional links to rationale and stakeholder involvement. As depicted in Fig. 2, the event server then customizes the generic message into a specific update directive for each dependent artifact.

As pointed out by Ramesh and Jarke [22], traceability links should be strongly typed in order to provide sufficient

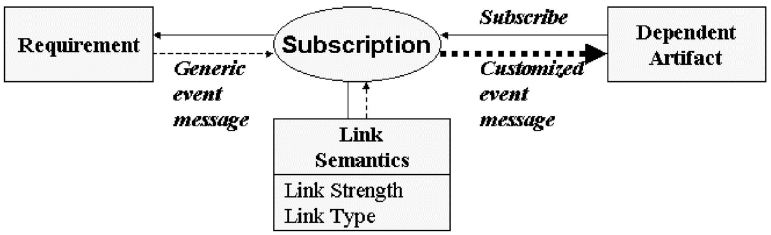

Fig. 2. Semantically rich event messages.

support for critical software engineering tasks. In EBT, this means that the published events carry appropriate task directives to provide guidance or support automation of the update process [60], [61]. In EBT, both the type and strength of the relationship are defined at the time the subscription is initially placed and can therefore be integrated into a customized event message for each dependent artifact. Link strength attributes are used to prioritize update tasks. At a higher managerial level, unresolved "strong" event messages are flagged as high-priority tasks, whereas lower strength messages are allowed to remain unresolved until artifacts are actually needed. A simple process guidance system [52] driven by the nature of the event, link characteristics, and artifact type can be embedded into the EBT event server in order to clearly define the types of actions required in response to a change. As an example, a generic *decompose* event message is published as:

> Requirements R17 | Decompose | 3 | R17, M-Net, \Functional \Whiteboard, The user shall be able to draw shapes on the whiteboard | R16, M-Net, \Functional\Whiteboard, The user shall be able to draw free hand shapes on the whiteboard using a drawing tool | R14, M-Net, \Functional\Whiteboard, The user shall be able to select predrawn shapes from a shape pallete to place on the whiteboard | Gaurav.Sethi | Tue Sep 25 12:10:09 PDT 2001.

When this message is sent to the dependent test case it is wrapped with the additional directive information: "Destination: 'TestCase-Draw shapes on whiteboard' | Priority: High | Directive: UpdateTestCase, RemoveLink, Re-executeTestCase, ReportResults". This message clearly defines the tasks that must be performed in order to restore the artifact to an updated state. From the perspective of the subscriber manager, all event messages must carry sufficient information to support the task of updating the impacted artifact and resolving links. This activity can be supported through the use of a design rationale system such as gIBIS [62], Sybil [63], [64], or Design Space Analysis (DSA) [65], [66], which record decision related factors such as issues, positions, arguments, and processes by which decisions are made. The current EBT prototype provides links to supporting documents such as meeting minutes and additional design documents; however, links to more extensive design rationales could easily be embedded into EBT event notification messages.

The message source field reflects the path taken by the current message. For example, if the message were initially sent to scenario S1 and then forwarded to a test case artifact, the message would contain the path data "Requirements R17/Scenario S1." This path data is crucial during the resolution of indirect artifacts if they have unresolved intermediate dependencies. A more thorough discussion of message forwarding is provided in Section 4.

3.4 Subscriber Manager

The subscriber manager is responsible for receiving event notifications and handling them in a manner appropriate to both the artifact being managed and the type of message received. A notification acceptance module receives event notifications, and a processing module supports the resolution of events as they occur [67]. The developer resolves events by updating the artifact and its associated links to reflect the specific change event that occurred. For example, if requirement $r1$ were decomposed into requirements $r2$ and $r3$ ($r1 \rightarrow r2 + r3$) and artifact a were dependent upon $r1$, then event resolution will involve determining whether new links should be established between a and $r2$ or between a and $r3$, and whether a should be modified in any way to reflect the change.

Decisions made concerning one event can impact other unresolved event notifications. For example if an event log contained the events $r1 \rightarrow r2 + r3$ followed by $r2 \rightarrow r4 + r5$ and, if during the resolution of the first event, the user determined that a was related to $r3$ but not to $r2$, then the second event might be irrelevant. The notification processor communicates with the event server to determine if artifact a has an explicit dependency upon requirement $r2$, or whether the event notification was sent as a precautionary measure in response to the initial decomposition of $r1 \rightarrow r2 + r3$. In the latter case, the second event notification is considered irrelevant and automatically removed from the event log. The event processor accomplishes these tasks by exchanging messages directly with the event server during the event resolution process.

4 Implementing Event-Based Traceability

When a change is introduced to a system the traceability method must support impact analysis of the proposed change in order to, among other things, avoid **identification errors**, as defined in Section 2.1. If the change is implemented, the traceability scheme must also support the updating of impacted artifacts and their related links in order to avoid **update errors**. Most traceability tools provide support for identifying impacted artifacts, but few provide management for the update process. This section describes EBT's support for both of these activities.

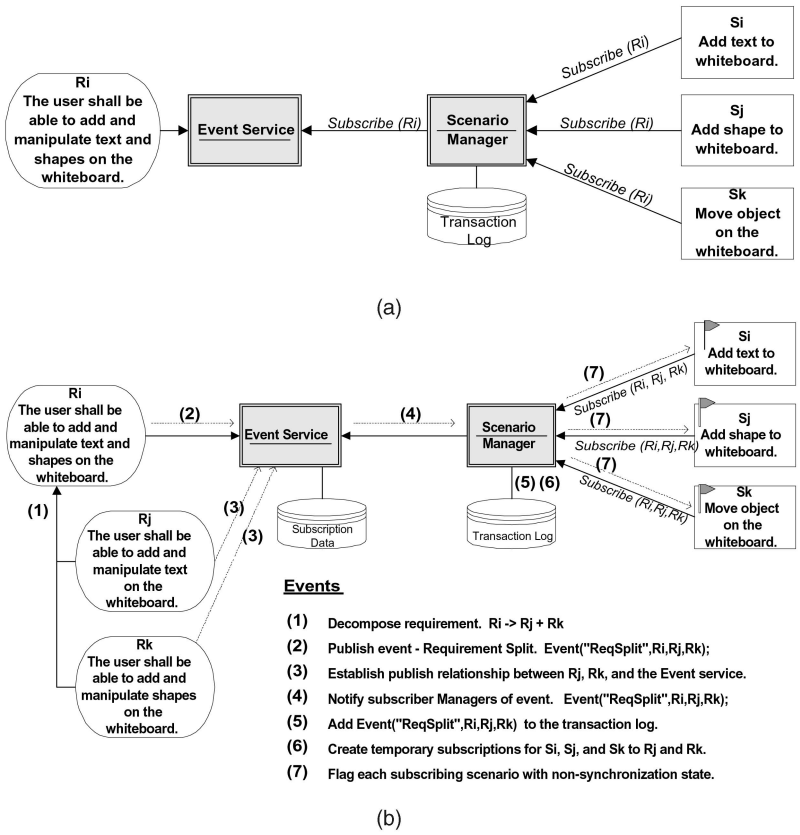

Fig. 3. EBT support for a requirements decomposition event. (a) Initial configuration using an Event Service. (b) Configuration following update.

4.1 Support for Artifact Identification

Artifact identification is crucial to higher-level activities such as impact analysis, requirements validation, and avoiding the introduction of unspecified features (aka gold-plating). To support the first two activities, the traceability infrastructure must provide the ability to trace forward from requirements to related artifacts such as design elements, test cases, and implemented code. To ensure that extraneous features are not introduced into a system in the later stages of development, the traceability infrastructure must support backward traceability from each artifact to the requirements from which it originated.

EBT supports both forward and backward traceability by use of a recursive query mechanism against the data in the event server. For example, to identify artifacts impacted by a change to a requirement r, the user can request a forward trace of all artifacts that subscribe to r either directly or indirectly. Similarly, to identify the set of requirements that specify the functionality or behavior of a given artifact a, the user can request a backward trace of all requirements to which a subscribes either directly or indirectly.

Artifact identification is therefore not dependent upon the event-based nature of the EBT scheme, but is based upon a simple query of the subscription database. The basic traceability features can therefore be offered without necessarily activating the event-notification mechanism, enabling a project manager to make project specific decisions about when activation should occur. In fact, it could be advantageous to activate different types of artifacts at different stages of the project. These decisions are determined on a project-by-project basis by considering project characteristics such as size, criticality, and volatility, and environmental factors such as distribution of the workplace, personnel skill level, and turnover rates. While EBT supports artifact **identification** and related impact analysis throughout the development process, it only supports artifact **update** activities once event notification has been activated.

4.2 Support for Artifact Maintenance

EBT supports updating artifacts and their related links, while simultaneously providing a project-level managerial visibility into the state of the system. As previously discussed, traditional traceability methods have a tendency to deteriorate, as time-pressured practitioners fail to systematically update impacted artifacts and links in a timely fashion. EBT directly alleviates the problem by providing an event based communication infrastructure that enables changes to be made and artifacts to be updated without unnecessarily requiring strict synchronization between owners of impacted artifacts. Change events are clearly documented in the individual event logs of each impacted artifact and can be resolved by the "owner" of each artifact in a timely fashion. The fact is that, in current practices, artifacts and links are frequently not updated in response to change, and EBT provides a managed solution to the problem rather than a dogmatic one.

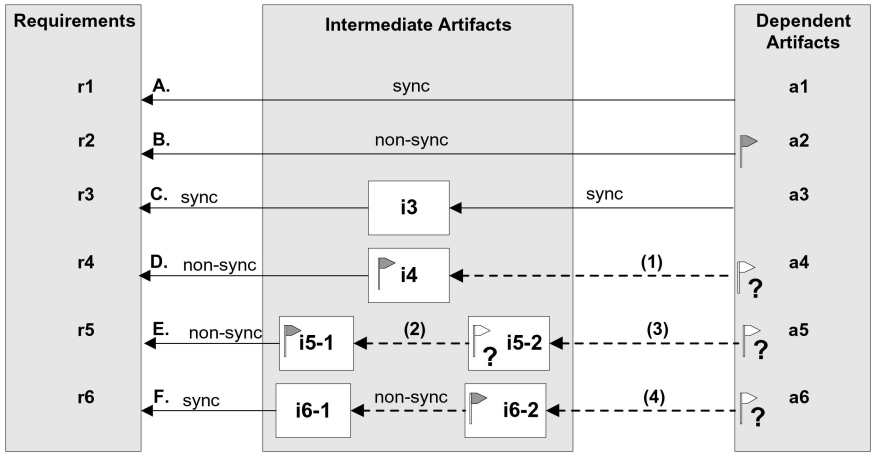

Fig. 4. Indirect dependencies.

To illustrate the application of EBT to requirements evolution, consider the example shown in Fig. 3a, Use-Case scenarios Si, Sj, and Sk are linked to requirement Ri via a subscription made by the scenario subscriber manager on behalf of each scenario. When the requirement is decomposed into two parts $\mathrm{R_j}$ and $\mathrm{R_k}$, a change event is automatically triggered and published to the event service. The event service looks up its internal database to identify subscribers and then notifies them of the event. The subscriber manager is responsible for handling different types of events according to the needs of the artifacts it manages. In this case, a *decompose* event requires manual intervention to resolve where new links might need to be established. The scenario subscriber manager updates its own transaction log to reflect the published event and flags all subscribers $(\mathrm{S_i}, \mathrm{S_j}, \mathrm{S_k})$ to indicate that they must synchronize their state with the transaction log before being used. The manager then updates the subscriptions of each subscriber so that they will be notified of events related to requirements $\mathrm{R_I}$, $\mathrm{R_j}$, and $\mathrm{R_k}$. This state of affairs is shown in Fig. 3b.

Critical artifacts should be updated in a timely manner, while less critical artifacts may remain nonsynchronized until they are needed in the future. At whatever point in time they are resynchronized, the developer uses the notification processor model with links to the rationale system to resolve event notifications and to update artifacts and their links accordingly.

This "just-in-time" scheme eliminates unnecessary overhead required to maintain all parts of the traceability scheme, yet, at the same time, provides the means to reconstruct links and update artifacts as needed. The project manager can view the state of event logs either by artifact type, criticality factor, link strength, functionality, or by owner and can therefore supervise the update activities and ensure that they are updated and maintained appropriately.

4.2.1 Indirect Dependencies

One of the major causes of failure in a traditional traceability scheme is the difficulty of recognizing and maintaining the numerous indirect links that exist between artifacts. In fact, in the case study introduced in Section 5 of this paper, a significant number of the dependencies represent indirect relationships such as links from test-cases to scenarios.

Fig. 4 depicts several different situations that can occur between artifacts. In case A, a simple dependency exists from $a1 \rightarrow r1$ with no synchronization problems: Both $a1$ and $r1$ have been updated. In B, a requirement $r2$ has been changed and the dependent artifact $a2$ flagged to reflect this nonsynchronized state. Before $a2$ can be used, its state must be synchronized. In C, an intermediate dependency exists within $a3 \rightarrow i3 \rightarrow r3$ but as all three items are synchronized there is no problem. The interesting cases occur in dependency paths D, E, and F. In each of these cases, the "**?**" and corresponding path sections labeled 1-4, depict unknown synchronization states.

In case D, an indirect dependency exists from $a4 => r4$, represented by the path $a4 \rightarrow i4 \rightarrow r4$. When a change event occurs that affects requirement $r4$, a change notification is sent to $i4$ and it is set in a nonsynchronous state to await change resolution. The problem is in determining what should happen to the dependent artifact $a4$ during this time period in which it is unknown how $a4$ will be impacted by the resolution of the change event in $i4$. Furthermore, this issue will not be resolved until $i4$ is synchronized and a subsequent event message forwarded to $i4$. Cases E and F exhibit similar problems, but are complicated even further by the presence of multiple intermediate artifacts.

There are two viable alternatives for handling this type of situation. Both of them are based upon the EBT premise that it is not necessary for all system artifacts to be concurrently synchronized at all times, but that each artifact must be able to ascertain its current state, and must know the steps that should be taken to restore itself to the desired state. In the first option, which we term "lazy notification," the indirectly dependent artifacts are not notified when one of their dependencies enters a nonsynchronized state. In the second option, which we term "pessimistic notification," all

dependents of the intermediate artifact are notified when it enters a nonsynchronized state. Both of these options are considered in the following section.

4.2.2 Lazy Notification

In lazy notification, dependent artifacts are not notified when an intermediate artifact enters a nonsynchronized state. This means that, prior to using an artifact, the state of all intermediate artifacts situated on dependency paths leading back to requirements must be tested. In a controlled environment, in which all traceable artifacts reside either on a single network, or on reliable processors exhibiting a high level of fault tolerance, this approach is viable. The critical issue is whether all intermediate artifacts would be available for querying as and when necessary. Lazy notification can also be successfully implemented if proxy artifacts are used to represent those artifacts that might not always be available; however, this introduces problems of duplication and synchronization.

The major benefit of lazy notification is the minimization of the number of notification messages; however, the major disadvantage is that the state of an individual artifact is not immediately apparent. This is not a significant problem on an individual scale because a developer can easily issue a query about the state of a specific artifact, but, in terms of general project management, it introduces an inhibitive level of overhead in the time required to obtain a system-wide managerial view. Lazy notification therefore degrades the ability of EBT to provide project level visibility.

4.2.3 Pessimistic Notification

Pessimistic notification requires all indirectly subscribed artifacts to be notified when a change occurs, even though it is not known whether the indirect subscribers will actually be impacted by the change or not. It pessimistically assumes that nonsynchronous states in intermediate artifacts will not necessarily be resolved in a timely fashion and that indirectly dependent artifacts should therefore be aware of their own potentially nonsynchronized state. This approach works best in finely grained traceability schemes in which event messages can be fine-tuned to small-grained artifacts or to event elements embedded within an artifact.

To actually implement a pessimistic notification scheme two additional change events are introduced. These additions are a "Non-Sync" message and a matching "Sync" message. The "Non-Sync" message is sent to all subscribers whenever the state of an intermediate artifact changes. Following resolution of the intermediate artifact, a "Sync" message and possibly a related event message are sent. These messages carry sufficient information describing how the original change event was resolved within the intermediate artifact to support the resolution algorithm in processing sequences of such events. The advantages of the pessimistic approach are that centralized proxies are not required and the state of any given artifact is clearly available through a local query, rather than through the global query needed to support lazy notification. Visualization techniques can be used to clearly represent the current state of each artifact in the traceability framework, and to define the needed to bring a given artifact to a synchronized state.

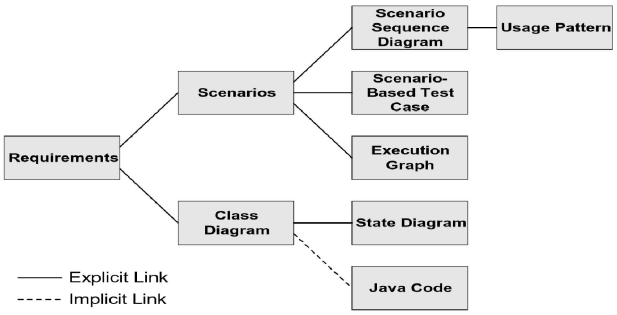

Fig. 5. Traceability paths between artifacts.

5 Applying EBT

In this section, we describe an example using M-Net, a web-based conferencing system with over 300 requirements and an initial set of 250 links [68], to illustrate the concepts described in this paper. M-Net artifacts include requirements, scenarios, class diagrams, scenario sequence diagrams, execution graphs, java code, and scenario based test cases. Project-level trace strategies established the linkable relationships between these artifacts as depicted in Fig. 5. The applied changes were designed to simulate the changes that might occur over a maintenance period of approximately 1-2 years. The artifacts impacted by the case-study included 16 requirements at level 0, 21 artifacts at level 1, 26 at level 2, and three at level 3.

5.1 Prototype

The EBT prototype was based upon a three-tiered architecture, consisting of three primary components including an event trigger, event service manager, and a subscriber manager. The event trigger was implemented on top of the DOORS requirements management system to manually capture change events as they occurred. The subscription interface was developed as a servlet to support the subscription process over the Internet. The event server receives published events at an assigned address, processes the events against the subscription files in order to identify subscribers, and forwards customized event notifications to the subscriber. The third component, known as the subscriber manager, was developed as a small server that receives event notifications on behalf of a particular type of dependent artifact. Events received from the event server are automatically resolved by the subscriber manager or stored in an event log for later semiautomated processing using a Change Processing Tool.

5.2 An Example of Change Maintenance in EBT

In this example, 20 change proposals were created, which translated into the creation of 46 new requirements, and

TABLE 1
Distribution of Messages

Type of Artifact	Path length	Total # msgs received	Avg # msgs per event log	Max # msgs per event log	Weighted avg # msgs per event log	Weighted max # msgs per event log
Class Diagram	1	35	3.5	7.0	3.5	7.0
Scenario	1	47	4.3	9.0	4.3	9.0
Scenario Sequence Diagram	2	30	6.0	10.0	12.0	20.0
Scenario Based Test cases	2	41	2.6	6.0	5.2	12.0
Usage Pattern	3	3	1.0	2.0	3.0	6.0
Other	n/a	3	1.0	1.0	n/a	n/a

publication of 159 event notifications. In this section, we describe one of the proposed changes in more detail and then provide a broader analysis of the impact of the remaining changes upon the system. The first change stated "Expand the chat room feature to display a picture of the current floor holder on the screen." In order to accomplish this, it was decided that the user should either provide a personal image or select a proxy image at the time of registration. The image of the current floor holder would then be displayed on the M-Net screen. To minimize bandwidth requirements, the images would be sent one time to all meeting members during the login phase.

The proposed change was found to impact the following four requirements: UD2: "The user shall be registered as a meeting member before being eligible to enter the M-Net meeting space;" UD20: "The user shall login prior to entering a meeting space;" FC5: "The chairperson shall assign the floor to any member of the meeting that has requested the floor;" and FC20: "The current floor holder's name shall be displayed for all meeting members to see."

5.2.1 Change Events

As a result, requirement UD2 was refined with the additional requirement UD15 stating that "The user shall register a self-image for display when that user holds the floor," reflecting the refinement of UD2 → UD2 + UD15. Requirement UD15 was then decomposed into three requirements stating that "The user shall be given the opportunity to supply a self image at the time of original registration" (UD16), "If the user does not supply an image, the user shall select a proxy image from the server" (UD17), and "The image shall be stored in a central repository" (UD19). The other requirements were refined, replace, or decomposed in a similar manner. As a result of these changes, the requirements manager published event notifications to the event server. The first event, Refine UD2 → UD2 + UD15, was published as:

> Requirements:UD2 | Refine | 1 | UD2, M-Net, \Functional\ UserData, The user shall be registered as a meeting member before being eligible to enter the M-Net meeting space.| UD15, M-Net, \Functional\ UserData, The user shall register a self-image for display when that user holds the floor.| RationaleLink | Fuhu Liu | Wed Sep 26 14:11:02 PDT 2001.

On receipt of this message, the event server identified a scenario (Scenario:Login) and class diagram (CentralServer:Login) as dependent artifacts. The event messages were then customized according to the initial subscription and forwarded to each dependent artifact. In this case, the Scenario:Login message was wrapped with the task definitions **"UpdateScenario | ResolveLinks,"** while the CentralServer:Login message was wrapped with the task definitions **"UpdateClassDiagram | ResolveLinks | UpdateCode | TestCode."** The inclusion of the code related tasks in the second message reflected the project-level decision to implicitly link class diagrams with the code that implements each class.

This message was then stored in the event logs of each of these artifacts to await resolution. At the same time, in conformance to the pessimistic notification scheme being implemented in the prototype, several "nonsync" messages were sent to artifacts dependent upon the Scenario:Login and CentralServer:Login. For example, the Scenario:Login forwarded a nonsync message to the two test case artifacts (TC:LoginValid and TC:Login Invalid) and to a performance related model (SSD:Login).

Table 1 depicts the average and maximum number of messages received by artifacts of each type following execution of all 20 proposed changes. These changes resulted in 64 messages sent to level 1 artifacts, 82 to level 2 artifacts, and 13 to level 3 artifacts. Code artifacts are not included in this table because according to the project-level strategies they were implicitly linked to class diagrams, and code related tasks were therefore forwarded as directives to impacted class diagrams. Path length depicts the distance of the artifact from requirements, and the "weighted" columns reflect the number of messages multiplied by the path length, which provides a metric for the effort required to resolve the events in the log.

5.2.2 Event Resolution

Event resolution is critical to the success of the EBT method. This section examines the event log of a scenario that captures the functionality of assigning floor control. The log is depicted in Fig. 6 following implementation of the 20 change proposals. As this artifact primarily exhibits level 1 dependencies upon requirements the event log consists of event messages sent directly from requirements.

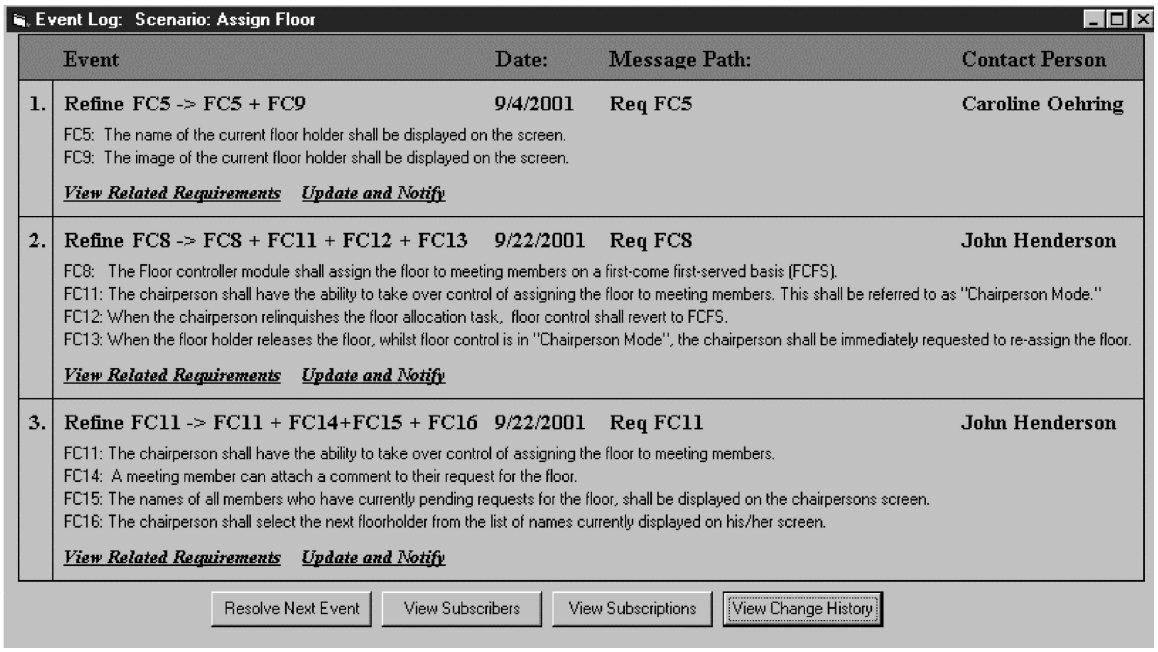

Fig. 6. Event log for a level one dependent artifact.

No indirect dependencies are involved and event resolution involves the following steps:

1. Update the "Assign Floor" Scenario representation to reflect the changes documented for each of these events.
2. Determine if the scenario should maintain its current traceability link to FC5, and/or if additional links should be created to FC9, FC11, FC12, FC13, FC14, FC15, and FC16. These types of decisions can be guided according to project level protocols concerning traceability granularity and capture strategies [20].
3. Compile a message describing changes made to the scenario and publish the message(s) to all dependent artifacts. The EBT event server handles the identification of subscribers and the mechanics of event notification, including resynchronizing the event logs of subscribers by replacing related nonsync events with the current notification.

Resolving event logs of artifacts with indirect links to requirements involves the additional steps of resolving intermediate artifacts. In EBT this is supported in one of two ways. First, critical intermediate artifacts tend to have "stronger" links to requirements, and their updates will therefore be prioritized and are likely to be resolved in a timely fashion. Second, a request from an indirectly linked artifact to resolve intermediate artifacts raises the priority level of related messages at those artifacts to the highest criticality level, therefore precipitating their resolution. EBT's project management tools provide comprehensive managerial support for maintaining critical artifacts in an updated state.

As depicted in Table 1, following the publication of the 159 messages, the longest event log of 10 messages was found in one of the scenario sequence diagrams [59]. In this case, the additional weight of having to resolve an intermediate dependency for each event, resulted in a weighted measure of 20 messages to resolve. In our experience with this and other similar examples, we found the resolution of such event logs supported by EBT's rationale documents and supporting change documentation to be no more difficult than if the updates had been applied immediately the requirements were changed. The benefits were the looser synchronization required by project participants, the inbuilt coordination mechanisms, and the ability to defer updating noncritical artifacts until such time as they were needed.

This example was applied to a nontrivial software system and simulated the effect of long-term change upon one major feature of the M-Net system. It demonstrated the feasibility of using EBT to solve synchronization problems related to updating artifacts and resolving traceability links. A longer-term empirical study applying EBT within a broader spectrum of applications is currently underway.

6 Conclusions

The major contribution of this work is to show how Event Based Traceability can be used to support the process of maintaining artifacts and their related links throughout the life of a software system. EBT directly addresses several of the identified causes of traceability failure, such as the problems related to the need for close coordination between team members, lack of visibility into the current state of the dependencies, lack of training, and the tendency of developers to fail to maintain links because of a perceived lack of immediate benefits. EBT reduces the need for strict synchronization between developers, while

providing a heightened level of visibility to project management. EBT therefore tackles problems related to maintaining and updating existing artifacts and links. One of the contributions of EBT is that it does not attempt to create a formal or a unified development environment, but provides a solution to the traceability update problem that can be implemented within the kind of heterogeneous and people-centric environment found in many current IT organizations and using popular tools such as DOORS and Rational Requsite Pro.

As long as traceability links are used strategically, EBT is scalable for use in larger systems. Even though EBT is based upon the scalable publish-subscribe paradigm, the number of event notification messages would likely become unmanageable if finely grained system-wide traceability links were established in a large system. However, it is widely recognized that traceability links should be established strategically according to critical success factors and should not be used in an excessive or haphazard approach. [20], [23], [60], [61]. Therefore, "large systems" and "unmanageable traceability links" need *not* be synonymous. The EBT architecture, which is based upon event notification, actually is more conducive to supporting a large number of changing relationships than are the more tightly coupled approaches such as matrices or hypertext links. In many other applications, such as news services subscriptions and hardware management, an event-based approach has already been proven to handle large numbers of subscriptions.

One area of future work will involve investigating the use of hybrid methods, in which dynamic event notification is used for only selective traceability paths, while other more stable paths are represented statically. In the example described in this paper, the starting point was an existing set of requirements and traceability links. We have considered, but not yet demonstrated how the approach would behave if applied to an entirely new development project. Further studies are needed to identify the factors that determine how and when event notification should be activated, although intuitively it would appear that activation should occur progressively as the project proceeds from inception through elaboration and construction.

In related work, we demonstrated the use of EBT to support performance related impact analysis through the use of speculative EBT events [59], [60], [61]. We are also currently investigating the critical role played by EBT in identifying requirements change events that introduce entirely new functionality, so that information retrieval type searches can be executed to search for impacted artifacts and establish new traceability links.

Our initial EBT work has focused upon the types of changes that occur within requirements, but change events in other types of artifacts such as scenarios could also be identified and defined as standard event messages. Ongoing research, including a longer-term empirical study is currently underway to address these open questions. A link to our EBT related work can be found at http://re.cti.depaul.edu/ebt.

ACKNOWLEDGMENTS

This research was partially funded by US National Science Foundation grant CCR-0098346. The authors would like to thank University of Illinois at Chicago students Gaurav Sethi, Fuhu Liu, Bin Wu, Arun Balakrishnan, and Kumar Jawaji, and DePaul students Haroon Chaudhry and Amit Uchat for developing the EBT prototype and supporting tools. The authors would like to thank Ugo Buy of the University of Illinois at Chicago and especially the anonymous reviewers for their invaluable comments and suggestions throughout the review process.

REFERENCES

[1] J.D. Palmer, "Traceability," *Software Requirements Eng.*, R.H. Thayer and M. Dorfman, eds., 1997.
[2] M. Jarke, "Requirements Traceability," *Comm. ACM*, vol. 41, no. 12, pp. 32-36, Dec. 1998.
[3] O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. First Int'l Conf. Requirements Eng.*, pp. 94-101, 1994.
[4] D. Zowghi and R. Offen, "A Logical Framework for Modeling and Reasoning about the Evolution of Requirements," *Proc. Third IEEE Symp. Requirements Eng.*, Jan. 1997.
[5] R.C. Sugden and M.R. Strens, "Strategies, Tactics, and Methods for Handling Change," *Proc. IEEE Symp. and Workshop Eng. of Computer Based Systems*, pp. 457-462, Mar. 1996.
[6] M.R. Strens and R.C. Sugden, "Change Analysis: A Step Towards Meeting the Challenge of Changing Requirements," *Proc. IEEE Symp. and Workshop Eng. of Computer Based Systems*, Mar. 1996.
[7] B. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, vol. 19, pp. 48-59, May 1973.
[8] R.K. Fjelstad and W.T. Hamlen, "Application Program Maintenance Study—Report to Our Respondents," technical report, IBM Corporation, DP Marketing Group, 1986.
[9] E. Bersoff, V. Henderson, and S. Siegel, *Software Configuration Management*. Prentice-Hall, 1980.
[10] P. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard, "LaSSIE: A Knowledge-Based Software Information System," *Comm. ACM*, vol. 34, no. 5, pp. 34-49, 1991.
[11] J. Martin and C. McClure, *Software Maintenance: The Problem and Its Solution*. Prentice-Hall, 1983.
[12] S. Dart, "Concepts in Configuration Management Systems," *Proc. Third Int'l Software Configuration Management Workshop*, pp. 1-18, June 1991.
[13] R. Pressman, *Software Engineering, A Practitioner's Approach*, fifth ed. McGraw-Hill, 2001.
[14] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenarios in System Development: Current Practice," *IEEE Software*, pp. 34-45, Mar./Apr. 1998.
[15] G. Antoniol, G. Casazza, and A. Cimitile, "Traceability Recovery by Modeling Programmer Behavior," *Proc. Seventh IEEE Working Conf. Reverse Eng.*, pp. 240-247, Nov. 2000.
[16] L. James, "Automatic Requirements Specification Update Processing from a Requirements Management Tool Perspective," *Proc. IEEE Workshop Eng. of Computer-Based Systems*, 1997.
[17] O. Gotel and A. Finkelstein, "Contribution Structures," *Proc. Second IEEE Int'l Symp. Requirements Eng.*, 1995.
[18] O. Gotel, "Contribution Structures for Requirements Engineering," PhD dissertation, Imperial College of Science, Technology, and Medicine, London, England, 1996.
[19] D. Leffingwell, "Calculating Your Return on Investment from More Effective Requirements Management," Rational Software Corporation, 1997. Available online at http://www.rational.com/products/whitepapers.
[20] R. Domges and K. Pohl, "Adapting Traceability Environments to Project Specific Needs," *Comm. ACM*, vol. 41, no. 12, pp. 55-62, 1998.

[21] A.M. Davis, *Software Requirements: Analysis and Specification.* Prentice-Hall, 1990.
[22] B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability," *IEEE Trans. Software Eng.,* vol. 27, no. 1, pp. 58-92, Jan. 2001.
[23] H. Kaindle, "The Missing Link in Requirements Engineering," *ACM SIGSOFT Software Eng. Notes,* vol. 18, no. 2, pp. 30-39, 1993.
[24] P. Kruchten, *The Rational Unified Process, An Introduction,* second ed. Addison-Wesley, 2000.
[25] M. Glinz, "A Lightweight Approach to Consistency of Scenarios and Class Models," *Proc. Fourth Int'l Conf. Requirements Eng.,* 2000.
[26] F.A.C. Pinheiro and J.A. Goguen, "An Object-Oriented Tool for Tracing Requirements," *IEEE Software,* vol. 13, no. 2, pp. 52-64, Mar. 1996.
[27] J. Cooke and R. Stone, "A Formal Development Framework and Its Use to Manage Software Production," *Tools and Techniques for Maintaining Traceability During Design, IEEE Colloquium, Computing and Control Division,* Professional Group C1 (Software Eng.), Digest Number: 1991/180, December 2, pp. 10/1 1991.
[28] "Draft Recommendation Z.URN: Languages for Telecommunication Applications—User Requirement Notation," Nov. 2000.
[29] E. Tryggeseth and O. Nytrø, "Dynamic Traceability Links Supported by a System Architecture Description," *Proc. IEEE Int'l Conf. Software Maintenance,* Oct. 1997.
[30] B. Azelborn, "Building a Better Traceability Matrix with DOORS," Telelogic INDOORS US, 2000.
[31] A. Lapeyre, "The Four Eras of Requirements Traceability," Telelogic INDOORS Europe, 1999.
[32] Rational Software Corporation, "Use Case Management with Rational Rose® and Rational RequisitePro®," available online at http://www.rational.com/products/reqpro/whitepapers.jsp, 2001.
[33] I. Spence and L. Probasco, "Traceability Strategies for Managing Requirements with Use Cases," Rational Software White Paper, 1999, http://www.rational.com/products/whitepapers/022701.jsp.
[34] "SE Tools Taxonomy–Requirements Traceability Tools," *Int'l Council Systems Eng.,* available online at http://www.incose.org/tools/tooltax.html, 2002.
[35] S. Gupta, J. Hartkopf, and S. Ramaswamy, "Event Notifier: A Pattern of Event Notification," *Java Report,* vol. 3, no. 7, 1998, available online at http://www.users.qwest.net/~hartkopf/notifier.
[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[37] D.L. Parnas, "On Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM,* vol. 14, no. 1, pp. 221-227, Apr. 1972.
[38] N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* vol. 14, no. 4, pp. 221-227, 1971.
[39] J. Cleland-Huang, "Robust Requirements Traceability for Supporting Evolutionary and Speculative Change," PhD dissertation, Univ. of Illinois at Chicago, Mar. 2002.
[40] "EvE, an Event-Driven Distributed Workflow Execution Engine,"available online at: http://www.ifi.unizh.ch/dbtg/Projects/EVE/eve.html, 1999.
[41] Geppert and D. Tombros, "Event-Based Distributed Workflow Execution with EVE," *Proc. iFiP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (MIDDLEWARE '98),* Sept. 1998.
[42] "TRAMS: Transactions and Active Database Mechanisms for Workflow Management," available online at http://www.ifi.unizh.ch/dbtg/Projects/TRAMs/trams.html, 2000.
[43] "SEAMAN: Software Engineering with Active Mechanisms," available online at http://www.ifi.unizh.ch/groups/dbtg/Projects/SEAMAN/, 1997.
[44] "SWORDIES: Swiss Workflow in Distributed Environments," available online at http://www.ifi.unizh.ch/dbtg/Projects/SWORDIES/index.html, 2000.
[45] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a Knowledge-Based Software Assistant," *Readings in Artificial Intelligence and Software Eng.,* C. Rich, and R. Waters, eds., Morgan Kaufmann, 1986.
[46] W.L. Johnson, M.S. Feather, and D.H. Harris, "The KBSA Requirements/Specification Facet: ARIES," *Proc. Sixth Knowledge-Based Software Eng. Conf.,* pp. 57-66, Sept. 1991.
[47] W.L. Johnson, M.S. Feather, and D.R. Harris, "Representation and Presentation of Requirements Knowledge," *IEEE Trans. Software Eng.,* vol. 18, no. 10, pp. 853-869, Oct. 1992.
[48] M. DeBellis, K. Miriyala, and W.C. Sasso, "Demonstration Description: The KBSA Concept Demonstration System," *Proc. Seventh Knowledge-Based Software Engineering Conference (KBSE '92),* W.L. Johnson, ed., Sept. 1992.
[49] M. DeBellis, K. Miriyala, S. Bhat, W.C. Sasso, and 0. Rambow, "Final Report: Knowledge-Based Software Assistant Concept Demonstration System," Technical report CDRL AO07, Government Contract F30602-89-C-0160, Oct. 1992.
[50] W.L. Johnson and M. Feature, "Building an Evolution Transformation Library," *IEEE Int'l Conf. Software Eng.,* pp. 238-248, 1990.
[51] V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing Process-Centered Software Engineering Environments," *ACM Trans. Software Eng. Methodology,* vol. 6, no. 3, pp. 283-328, 1998.
[52] K. Pohl, K. Weidenhaupt, R. Domges, P. Haumer, M. Jarke, and R. Klamma, "PRIME–Toward Process-Integrated Modeling Environments," *ACM Trans. Software Eng. and Methodology,* vol. 8, no. 4, pp. 343-410, Oct. 1999.
[53] M. Dowson, "Consistency Maintenance in Process Sensitive Environments," *Proc. Process Sensitive Software Eng. Architectures Workshop,* Sept. 1992.
[54] K. Pohl, *Process Centered Requirements Engineering.* John Wiley and Sons Ltd., 1996.
[55] K. Pohl and K. Weidenhaupt, "A Contextual Approach for Process-Integrated Tools," *Proc. Sixth European Software Eng. Conf. (ESEC/FSE '97),* M. Jazayeri and H. Schauer, eds., 1997.
[56] J. Cleland-Huang, C.K. Chang, and Y. Ge, "Supporting Event Based Traceability through High-Level Recognition of Change Events," *IEEE Proc. Int'l Computer Software and Applications Conf. (COMPSAC),* Aug. 2002.
[57] D. Barnard, G. Clarke, and N. Duncan, "Tree-to-Tree Correction for Document Trees," Technical Report 95-372, Dept. of Computing and Information Science, Queen's Univ., Kingston, Ontario, Canada, 1995, available online at http://www.cs.queensu.ca/TechReports/ Reports/1995-372.pdf.
[58] S. Selkow, "The Tree-to-Tree Editing Problem," *Information Processing Letters,* vol. 6, no. 6, pp. 184-186, Dec. 1977.
[59] J. Cleland-Huang, C.K. Chang, H. Kim, and A. Balakrishnan, "Requirements Based Dynamic Metrics in Object-Oriented Systems," *Proc. Fifth IEEE Int'l Symp. Requirements Eng.,* pp. 212-219, Aug. 2001.
[60] J. Cleland-Huang, C.K. Chang, H. Hu, J. Kumar, G. Sethi, and J. Xia, "Requirements Driven Impact Analysis of System Performance," *Proc. IEEE Joint Requirements Eng. Conf.,* pp. 289-296, Sept. 2002.
[61] J. Cleland-Huang, C.K. Chang, and J. Wise, "Automating Performance Related Impact Analysis through Event Based Traceability," to be published.
[62] J. Conklin and M.L. Begeman, "gIBIS: A Tool for All Reasons," *J. Am. Soc. for Information Science,* pp. 200-213, May 1989.
[63] J. Lee, "SIBYL: A Qualitative Decision Management System," *Artificial Intelligence at MIT: Expanding Frontiers,* P. Winston and S. Shellard, eds., pp. 105-133, 1990.
[64] J. Lee, "SIBYL: A Tool for Managing Group Decision Rationale," *Proc. Computer Supported Cooperative Work (CSCW '90),* 1990.
[65] A. MacLean, R.M. Young, and T.P. Moran, "Design Rationale: The Argument Behind the Artifact," *Proc. Human Factors in Computing Systems (CHI '89),* 1989.
[66] A. MacLean, R.M. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction,* vol. 6, nos. 3 and 4, pp. 201-250, 1991.
[67] F. Liu, "Notification Processing—Event Based Traceability," Master Project Report, Univ. of Illinois at Chicago, Mar. 2001.
[68] "M-Net: Meeting Net Online Demo,"available at http://icse.cs.iastate.edu, 2000.
[69] "EBT Online Demo,"available at http://re.cs.depaul.edu, 2003.

Jane Cleland-Huang received the MS degree in computer science from Governors State University, Illinois, in 1998, and the PhD degree in computer science from the University of Illinois at Chicago in 2002. She is currently an assistant professor in the Department of Computer Science, Telecommunications, and Information Systems at DePaul University, Chicago. Her research interests include requirements engineering, traceability, and software development processes. She is the coauthor of the book *Software by Numbers: Strategies for Low-Risk High-Return Development* (Prentice-Hall, 2003). She is a member of the IEEE Computer Society and the ACM.

Carl K. Chang is a professor and chair of the Department of Computer Science at Iowa State University. His research interests include requirements engineering, software architecture, and net-centric computing. He is a founding member of the IEEE International Requirements Engineering Conference (RE). He currently serves on the steering committee of RE and as general chair of RE '03. He is also the chair-elect of the steering committee for the IEEE-CS/IPSJ International Symposium on Applications and the Internet (SAINT) after serving as the program chair of SAINT2002 and general chair of SAINT2003. He is also active in the educational activities and spearheaded the Computing Curricula 2001 (CC2001) project jointly sponsored by the IEEE Computer Society, ACM, and the US National Science Foundation. He served as the editor-in-chief for *IEEE Software* in 1991-94. He is a fellow of IEEE and the future (2004) president of IEEE Computer Society.

Mark Christensen received the BS degree in physics and mathematics from Wayne State University and the MS degree in physics from Purdue, where he was a Woodrow Wilson Fellow. His doctorate from Wayne State is in probability theory. He is currently an independent consultant based in St. Charles, Illinois. He serves a national client base, offering process and project evaluation services, and project management training. His customers include industrial, governmental, and academic organizations. From 1988 through 1994, he was the director of software engineering for the Defense Systems Division of Northrop Grumman Corporation and, from 1996 through 1998, he served as the vice president of Engineering of the Electronic Systems Division of Northrop Grumman Corporation in Rolling Meadows, Illinois. Prior to this, he was an associate professor of mathematics at the Georgia Institute of Technology, where he worked in the areas of computational probability, numerical methods, and computer graphics and simulation. Dr. Christensen chairs the press operations committee of the IEEE Computer Society. He is the coauthor with Dr. Richard Thayer of the book *The Manager's Guide to Software Engineering's Best Practices* (IEEE Computer Society, 2002). He is also the author of articles on software construction and software professionalism used in the preparation guide for the IEEE Certified Software Development Professional program. He has authored more than 20 papers in refereed mathematical and software journals.