A Scenario-Driven Approach to Traceability

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90292
+1 310 578 5350
aegyed@teknowledge.com

ABSTRACT
Design traceability has been widely recognized as being an integral aspect of software development. In the past years this fact has been amplified due to the increased use of legacy systems and COTS (commercial-off-the-shelf) components mixed with the growing use of elaborate "upstream" software modeling techniques such as the Unified Modeling Language (UML). The more intensive emphasis on upstream (non-programming) software development issues has, however, widened the gap between software components (e.g., subsystems, modules) and software models (e.g., class diagrams, data flow diagrams), creating the need for a better understanding of the intricacies and interrelationships between both of them. This paper demonstrates how observable run-time information of software systems can be used to detect traceability information between software systems and their models. We do this by employing a technique that evaluates the "footprints" that usage scenarios (e.g., test cases) make during the execution of software systems. Those footprints can be compared, resulting in additional traceability information among modeling elements associated with those scenarios. Our approach is tool supported.

Keywords
traceability, test scenarios, models, UML, legacy systems

1 INTRODUCTION

Science and engineering alike stress the importance of being able to reproduce results. A general technique for enabling this is being able to *trace* ones steps from inception to transition. If done comprehensively, tracing can outline every step along the way of how a problem is transformed into a solution, including intermediate results and findings. Software development needs traceability for that same reason [15].

It has been argued repeatedly that software developers need to capture traces [6,17] between requirements, design, and code. Most trace capture methods, however, require extensive manual intervention [7,9] (e.g., via naming dictionaries or traceability matrices), only rare cases have (semi) automated support [2,12] (e.g., via formal methods). Despite the effort spent in generating and validating trace information, there is often only little trust in them since they may have (potentially) become obsolete due to separate evolution of models and systems [11]. This causes a dilemma because if a software analyst, architect, designer, or programmer cannot trust existing trace information, then it has lost its value [6,7]. It does not matter whether, say, 70% of the trace information is actually correct if it is unknown which 70% that is. The mistrust in the accuracy of models and their trace information leads to a practice of disregarding them [3], frequently only creating them because some process requires to do so [5,9]. This, however, reduces models and diagrams to mere pictures without any "life."

The recent emergence of new modeling techniques, like the Unified Modeling Language (UML) [13], and their rapid and increasing acceptance into the main stream software development process, has revitalized the need for understanding the intricacies and interdependencies between model elements and their corresponding software systems. This is because models promise one very important thing: a different way of looking at software and, in doing so, a better way of gaining insight into it. Nevertheless, the past years has only seen little progress in the problem of how to relate models and systems. Without such relationships one may question the foundation of modeling since there is little (if no) value in using a software model that does not consistently represent the real software system [4,6]. Thus, what is needed is trace information and the lack of it constitutes a severe problem – also known as the traceability problem [6,8].

This work introduces a new, strongly iterative approach on how to generate traceability information based on observing test scenarios that are executed on running software systems. These observations are then used for establishing traces between model elements (e.g., classes and data flow elements) and their corresponding source code (the system). Furthermore, our approach then shows how to generate trace information between model elements themselves (e.g., model elements of different diagrams). The latter is done by comparing the impact that test scenarios have on the code level. We call that impact "footprint."

As a pre-requisite, our approach requires (1) the existence of an observable and executable software system, (2) a cor-

Proceedings of the
23rd International Conference on Software Engineering
Toronto, Canada, May 2001, pp. 123-132.
(ICSE 2001)

responding software model, and (3) scenarios describing test cases or usage scenarios of the software system or its components. Our approach first creates trace information between the running system and scenarios followed by comparing those traces with hypothesized traces. Our approach then generates new trace information and validates existing ones. We will demonstrate the workings of our approach in context of the Inter-Library Loan (ILL) system [1], a third-party software system. This example also incorporates the use of a COTS component (Microsoft Access®) demonstrating our approach's ability to also support third-party software components that are not readily observable.

Our approach may be used during reverse engineering (e.g., of recently developed systems or legacy code) or during forward engineering. The latter may not be intuitive at first since a running system is required, however, partial implementations (e.g., sub components) or even prototypes are sufficient to commence analysis. Our tool, called TraceAnalyzer, automates our approach.

2 APPROACH OVERVIEW

Our approach adds a reengineering element towards trace information detection and validation. It does this by matching development models (static information) against the actual implementation and execution of the software product (dynamic information) in order to identify dependencies between them. These dependencies can then be used to establish traces. Our approach works once an executable or simulateable software system becomes available which may not necessarily be the final release of a system but could also be a partial implementation or incremental prototype.

Once an executable system is found, the behavior of that system is observed using kinds of test scenarios that are typically defined during its development (e.g. acceptance test scenarios, module test cases, etc.). By applying those scenarios onto the system the internal activities of that system can be observed and recorded. Since those observations correspond directly to scenarios this implies that trace information are found between those scenarios and the system – we refer to the source code that is executed while testing a scenario as *footprint*. Our trace analysis approach relies on monitoring tools for spying into software systems during their execution or simulation. Those tools are readily available. For instance, we used a commercial tool from Rational Software called Rational PureCoverage® in order to monitor the running Inter-Library Loan (ILL) system. As such, the tool monitored what lines of code were executed how many times and when.

Scenarios, which are tested against the running software system, can be of different forms. Scenario descriptions may be in plain English, in some form of diagram (e.g., sequence diagram [13]), or in a formal representation that can be interpreted by an automated testing tool. It is not important for our approach whether scenarios are tested manually or automatically, as long as a trace observation tool, like Rational PureCoverage, is used to observe the footprints these scenarios make. Testing scenarios may thus be as simple as manually interpreting them or as advanced as automatically testing them via testing tools. Either way, this paper contributes an automated approach for interpreting observed run-time information to establish traces between model elements. In particular we are able to generate and validate the following four types of traces (see also Figure 1):

(a) traces between scenarios and system,
(b) traces between model elements and system,
(c) traces between scenarios and model elements, and
(d) traces between model elements

Our approach consists of four major activities called *Hypothesizing*, *Atomizing*, *Generalizing*, and *Refining*. *Hypothesizing* (the first activity) requires reasoning about traces that may exist in the analyzed system (traces of types 'b', 'c' or 'd'). Hypothesized traces can often be elicited from system documentations or corresponding models. If no documentation is available, hypothesizing may have to be done manually, however, hypothesizing becomes more automated over time since traces generated via an iteration of our approach can be used as hypothesized traces in a successive iteration. Our approach uses current knowledge about traces and permutates them with observable footprints (trace of type 'a') to build a footprint graph (see *atomizing*). This footprint graph becomes the foundation for advanced traceability reasoning in *generalizing* and *refining*. Hypothesized trace information does not have to be comprehensive nor does it have to be (fully) correct. We will show examples that demonstrate that inconsistent and insufficient trace hypotheses may result in contradictions and ambiguities during *generalizing* and *refining* that make them detectable.

The second activity, *Atomizing*, builds a so-called *footprint graph* out of observable trace information. The goal of the footprint graph is to depict the most basic (atomic) footprints any two scenarios have in common. The footprint graph is then used as a foundation for the two remaining activities *Generalizing* and *Refining*. *Generalizing* traverses the footprint graph starting at its "leaves" (most atomic footprint elements) to propagate (generalize) trace information to their "parents." *Refining*, the reverse activity, then

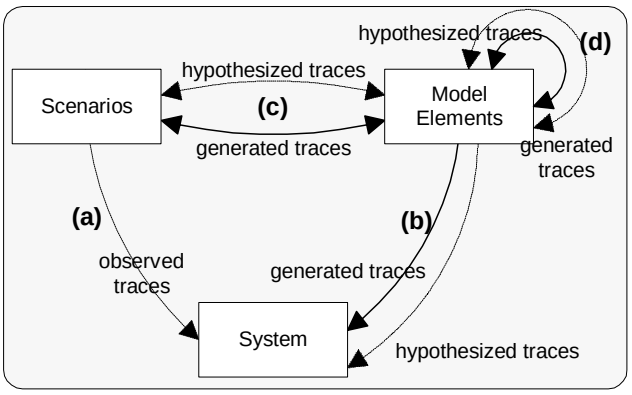

Figure 1. Trace Types and their Interrelationships

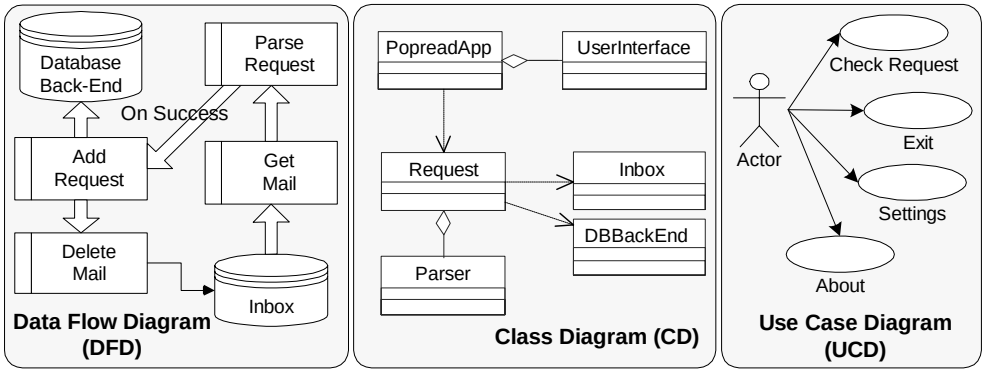

Figure 2. Popread Component of ILL System [9] represented in three different diagrams

traverses the footprint graph starting at its "roots" (most general parents) to propagate (refine) trace information to their leaves.

Our approach can generate new trace information of types (b), (c), and (d) (recall Figure 1) or, if contradictions are encountered, our approach can also invalidate existing (hypothesized) trace information. Since initial traces are hypothesized, contradictions imply situations where the hypothesis does not hold. The amount of hypothesized traces required to use our approach is entirely variable. As a general rules, the more correctly hypothesized trace information is available the less contradictions are encountered and the less ambiguous are found.

3 EXAMPLE

To evaluate our approach, we applied it to some real-world projects. This paper shows one of those projects, the Inter-Library Loan System (ILL) [9], to demonstrate the workings of our approach. The ILL system automates the lending process of books, papers, and other forms of media between libraries. The system was successfully transitioned to the customer where patrons use it to order documents not available locally. ILL allows document requests to be filed via a web browser which in turn submits them as tagged emails to a specially dedicated email account. There, email messages get read and processed by a part of the ILL system called the *PopRead* component.

For simplicity, this paper only uses the *PopRead* component which is illustrated in Figure 2. On the left, the figure shows the functional decomposition of the *PopRead* component in form of a data flow diagram (DFD) (as depicted in their *Software System Requirements Document* (SSRD) [9]). In the middle, the figure shows the corresponding object-oriented design using a UML class diagram, and, finally, on the right, the figure shows a use case diagram depicting the major services the *Popread* component has to offer (from a user interface perspective). All three views show the *PopRead* component in a high-level fashion. It can be seen that *PopRead* reads email messages from the *Inbox* (POP3 account), parses them, and if the parsing is successful (OnSuccess), stores the identified requests on a back-end database (MS Access®).

Table 1. Some ILL Test Scenarios

A	Show the About box
B	Check for new mail without any present
C	Check and unsuccessfully read one mail
D	Modify settings
E	Startup and shutdown
F	Startup, show about, and shutdown
G	Check Request
H	Delete inbox entry

Associated with the *PopRead* component, the ILL developers also created a number of usage scenarios (Table 1). One such scenario is *checking for mail and unsuccessful parsing* (scenario C). Another scenario is the *starting of the Popread application, displaying its About box, and shutting it down* (scenario F). It was part of the development team's process to create structural views (such as the ones in Figure 2) as well as behavioral views (such as the scenarios in Table 1). The ILL team used mostly UML sequence diagrams to describe scenarios and UML class diagrams and data flow diagrams to describe structure.

Despite all the modeling and documentation, the ILL team failed to describe how the *PopRead* design elements (Figure 2) were actually implemented. Thus, trace information from the dataflow, class, and use case diagrams to the source code (the software system) is missing. It remains unclear from the specification provided, what source code functions and implementation classes are used by what higher-level elements (e.g., what lines of code are used by the dataflow element *Add Request*). Additionally, the development team did not specify how the dataflow, class, and use case diagrams relate to one another. It remains unclear what higher-level classes make up individual data flow processes (e.g., is the class *Request* used in the data flow diagram?). Even in this rather small example those traces are not obvious since functional and object-oriented decomposition is mixed.

Some documents, however, did capture trace information. The only problem is that we cannot safely assume their correctness. We already discussed previously that trace information is frequently defined and maintained explicitly (e.g., in documents) which requires manual labor in keeping them up to date [2]. Even if the development team took the effort to document the final traces, the activity of identifying trace information manually would still be very error-prone. In the following, we will show how we can use existing trace information, despite the lack of faith in them, to generate new ones and validate old ones.

4 HYPOTHESIZING

Testing the system or some of its components and observing their traces is a straightforward activity. Table 2 shows the summary of observing the footprints of the eight test

Table 2. Observeable Scenario Footprints

Class ↓ Scenario →		A	B	C	D	E	F	G	H
CAboutDlg	0	10					10		
CILLDB	1			3	2	2	2	4	
CILLDBSet	2			4				4	
mailreader	3		3	4	7	1	1	5	1
parsing	4			6				6	
POP3	5		8	10	2	2	2	11	1
CPOP3Dlg	6		4	4	1	4	4	4	
CAPP	7					1	1		
CMainWin	8	3	5	5	3	6	7	5	
CSettingsDlg	9				15				

scenarios from Table 1 using the Rational PureCoverage tool. With a footprint we mean the classes that were executed while testing a scenario. The numbers in Table 2 indicate how many methods of each class were used. For instance, scenario "A" used ten methods of the class *CAboutDlg* and three methods of the class *CSettingsDlg*. Table 2 does not display the actual methods to reduce the complexity of this example (the footprint graph shown later would otherwise get too big). Nevertheless, by only using classes the generated traces will still be useful, albeit, course-grained. If more fine-grained traces are needed then trace analysis has to be performed by observing the methods of classes or even the lines of code. Our approach remains the same.

Hypothesizing is the only manual activity of our approach. The goal of this activity is to reason (hypothesize) about potential trace information. One potential trace could be from scenario A to the use case *About* in Figure 2 (see also Table 4). Another potential trace could be from the high-level class *PopreadApp* to the implementation classes *CApp* and *CMainWin* (see also Table 3). Naturally, our approach requires only a fairly limited amount of hypothesized trace information; otherwise, the cost of using it would be too high. Table 4 and Table 3 show a list of twelve traces we hypothesized. We assume that most traces are at least partially correct, however, our approach can pinpoint wrong traces plus create new ones by matching the derived trace information against the observed trace information. These traces, together with the trace observations that were automatically generated (Table 2) can now be used for further reasoning.

Table 4. Hypothesized Trace Information

Scenario	*Model Elements*
A	UC::About [p]
B	DFD::Inbox, DFD::GetMail [g/h]
C	CD::Request, CD::Inbox, CD::Parser [c/d/f]
D	UC::Settings [o]
E	CD::PopreadApp [a]
F	CD::PopreadApp, CD::UserInterface [a/b]
G	UC::CheckRequest, DFD::Inbox, DFD::GetMail, DFD::ParseRequest, DFD::DeleteMail, DFD::DatabaseBackEnd [m/g/h/i/j/k/l]
H	CD::Inbox [d]

Table 3. Hypothesized Trace Information

Model Elements	*System*
CD::Request [c]	mailreader, POP3 {3/5}
CD::DBBackEnd [e]	CILLDB, CILLDBSet, POP3 {1/2/5}
CD::PopreadApp [a]	CApp, CMainWin {7/8}
CD::UserInterface [b]	CAboutDlg, CPOP3Dlg, CSettingsDlg {0/6/9}

5 ATOMIZING

In order to reason about traces and how they relate to model elements, we need to intertwine scenarios, model elements, their footprints, and hypothesized trace information. In order to do this, we have devised a *footprint graph*. Figure 3 depicts the complete footprint graph for the eight scenarios in Table 1[1]. This graph also forms the foundation for the remaining activities of our approach: *Generalizing* and *Refining*.

One property that graph has is that footprints of observed trace information is split up into as many nodes needed to explicitly represent all possible overlaps between scenarios (overlaps are footprints that any two scenarios have in common). For instance, scenario "A" has the observed footprints {0/8} (Table 2) and, similarly, scenario "B" has the observed footprints {3/5/6/8} (see also nodes "A" and "B" in Figure 3). However, both scenarios also overlap since they share the footprint {8} which corresponds to the implementation class *CMainWin* (Table 2). In order to capture that overlap, another node was created (node "AB") and that node was then declared "child" of both parents (nodes "A" and "B"). Nodes "A" and "B" also have footprints they do not share but those are not of importance unless they overlap with footprints of yet other scenarios.

Once scenario "C" is added to the footprint graph, it is found that it overlaps with scenarios "A" and "B." In fact, the footprint of scenario "B" is a subset of the footprint of scenario "C." The node for scenario "B" is thus made into a child of the node for "C" (called "C/G" in Figure 3). In rare cases it may even happen that two scenarios have the exact same footprint as in the case of scenarios "C" and "G." A common node for both "C" and "G" was thus created, ergo its name "C/G." Note that scenario "C" also overlaps with scenario "A," however, that overlap was already captured via its link to node "B" which, in turn, has node "AB" as one of its children. Therefore, another property of our graph is that overlaps between scenario footprints are captured in a hierarchical manner to minimize the amount of nodes required.

Building a footprint graph is not very difficult since it only involves two major steps. For each new scenario added to

[1] In order to make the figure readable, the graph is depicted in a tree structure using duplication (e.g., the node B is replicated several times but should be considered one node).

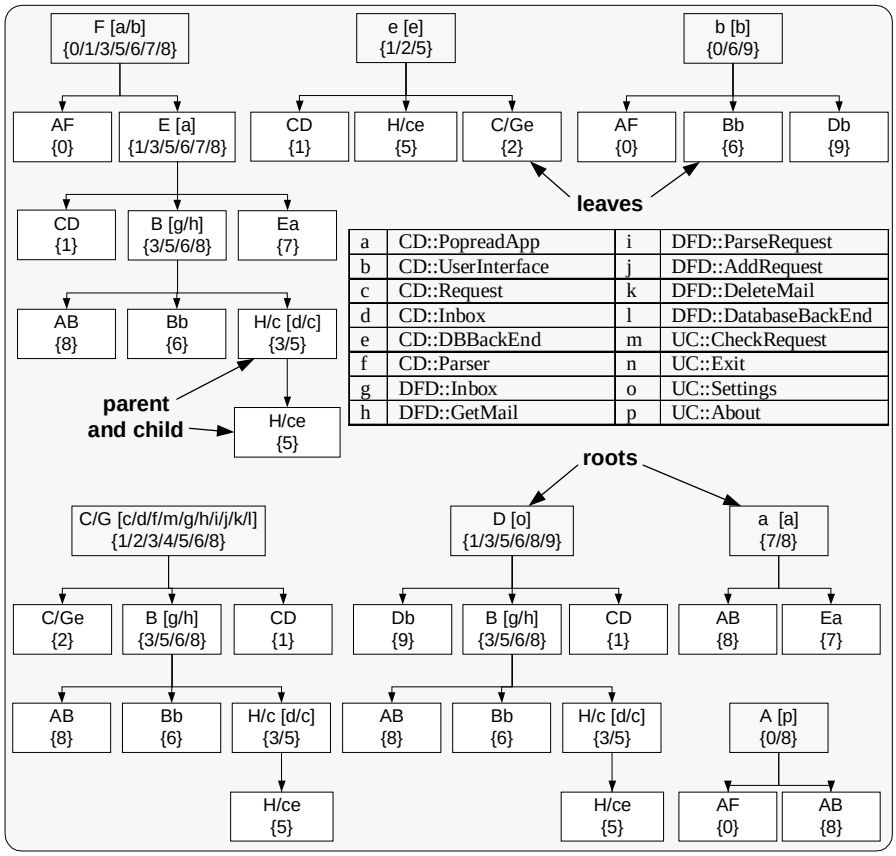

Figure 3. Footprint Graph (in tree shape) representing atomic footprints

the graph, the first step tries to identify the largest node (large with respect to the amount of footprints covered) that is either equal or part of the new scenario (e.g., as "C" was equal to "G" or as "B" was a part of "C"). If two nodes are equal then they get merged together and if one node is a subset of the other then a parent-child relationship is established.

In case no node is found or in case a found node only partially covers the footprint of the new scenario, a second step is performed on the remaining footprints of the new scenario. The second step tries to find the smallest nodes (small with respect to the amount of footprints covered) that overlap with the new scenario (e.g., scenario "A" overlapped with "B"). In this case a new node is created containing the overlapping footprints. That node is then declared child of both overlapping parents (e.g., node "AB").

Applying above algorithm to all scenarios results in a footprint graph like Figure 3. The naming convention used in the graph is that each node refers to the scenarios it was created from. In case a single letter is used (e.g., "B") the node refers to the complete scenario "B." Accordingly, the footprints listed in curly brackets must be equal to the footprints the scenario relates to (in Table 2 it can be seen that scenario B relates to footprints 3, 5, 6, and 8 just like node "B" does). In case a node has multiple letters separated by a slash then their corresponding scenarios have equal footprints (e.g., "C/G"). In case a node has multiple letters without any separators then it represents a subset of footprints its parent nodes have in common (e.g., node "AB" contains the overlapping footprints of nodes "A" and "B").

The graph also makes use of lower case and upper case letters. The upper case letters refer to scenarios that have observed trace information (like the scenarios in Table 2) whereas the lower case letters refer to model elements that have (hypothesized) trace information (like the model elements in Table 3). The key on how to interpret the lower case letters is given in Figure 3. For instance, node "a" stands for the class *PopreadApp* (*CD::PopreadApp*), which, in Table 3, was hypothesized to trace to *CApp* {7} and *CMainWin* {8} (note: "CD" stands for class diagram). A final convention used in the footprint graph is its references to model elements. For instance, node "B" was hypothesized to refer to *DFD::Inbox* [g] and *DFD::GetMail* [h] (Table 4), ergo "[g/h]." We use these naming conventions for brevity since otherwise Figure 3 would be too crowded and might not fit into a single page. Naturally, using abbreviations is only useful for demonstration purposes.

Another important property of the footprint graph is that children of nodes (e.g., "ABa" and "ABb" for node "AB") have non-overlapping footprints; each child represents a distinct subset of its parent's footprint. Also, the union of all children's footprints is always equal or a subset of the footprints of their respective parents. These two properties are important for the next two steps where we use the footprint graph for *Generalizing* and *Refining*. Although, *atomizing* is mainly a preparation step, we can already derive some new trace information out of it. For instance, node "C/G" shows a case where two scenarios share the exact same footprints. Through Table 4 we know that scenario C was hypothesized to trace to *CD::Request*, *CD::Inbox*, and *CD::Parser* [c/d/f]; and that scenario H was hypothesized to trace to *UC::CheckRequest*, *DFD::Inbox*, *DFD::GetMail*, *DFD::ParseRequest*, *DFD::DeleteMail*; *DFD::DatabaseBackEnd* [m/g/h/i/j/k/l]. We can now make three trace assumptions:

1. The use case *CheckRequest* relates to the classes {*Request*, *Inbox*, *Parser*} (trace type (d) in Figure 1)
2. The use case *CheckRequest* also relates to the data flow elements {*Inbox*, *GetMail*, *ParseRequest*, *DeleteMail*, *DatabaseBackEnd*} (trace type (d))
3. The classes {*Request*, *Inbox*, *Parser*} relate to the data flow elements {*Inbox*, *GetMail*, *ParseRequest*, *DeleteMail*, *DatabaseBackEnd*} (trace type (d))

4. The scenarios "C" and "G" trace to various additional model elements (trace type (c) in Figure 1)

The rationale behind this reasoning is that if multiple model elements share the same lines of code (footprint) then they have to be related. For instance, the first assumption above implies that if *CheckRequest* executes the same lines of code as *Request*, *Inbox*, and *Parser* do then, on a higher-level, it can be said that *CheckRequest* uses the classes *Request*, *Inbox, and Parser*. There is only one significant assumption in above observation: it is assumed that the given (hypothesized) trace information is correct (consistency assumption). This statement seems to contradict an earlier statement where we claimed that hypothesized information need not always be correct. In fact, both statements are accurate. In Table 4 and Table 3 we made hypotheses we *assume* to be correct. Therefore, this approach will treat them as being correct (even if they are not) until a point of contradiction is reached that invalidates them. We can already find one such contradiction in our footprint graph.

Node "H/c" shows an example where scenario "H" and model element "c" were found to have the same footprints (see Table 2 and Table 3). Naturally, node "H/c" must trace to model element [c] since that element is part of the node, however, node "H/c" also traces to the model element [d] (*CD::Inbox*) as defined in Table 4. In that respect, this case is similar to the previous case "C/G," although here is one fundamental difference: It was observed that *deleting an inbox entry* (scenario "H") results in the same footprint as the class *Request* was hypothesized to have (model element "c"). Making again a consistency assumption (as we did above) we have to conclude that model element [c] must trace to the model elements of scenario "H" since they share the same footprint. This implies that model element [c] must trace to model element [d] – a contradiction. The contradiction implies that either trace information is incorrect (e.g., scenario "H" should also trace to *Request*) or it implies that class *Inbox* is used by class *Request*. In this case, the latter is true as can be seen in Figure 2.

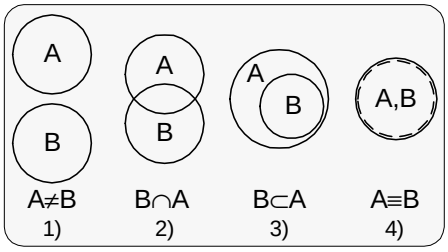

Figure 4. Possible overlaps of footprints

To generalize from above cases, Figure 4 depicts all patterns that may be encountered while comparing any two footprints. Those patterns also form the foundation in understanding the relationships between model elements:

1) If the footprints of two scenarios do not overlap then this implies that there is no relationship between both scenarios and their model elements
2) If the footprints of two scenarios partially overlap then this implies that some common model elements exist
3) If the footprint of one scenario is completely engulfed by the footprint of another one then this implies that model elements of the one are also used by the other
4) If the footprints of two scenarios are exactly alike then this implies the equivalence of both scenarios in the model elements they use

6 GENERALIZING

The pure hierarchical decomposition of the footprint graph allows us to further reason about the dependencies between its nodes. Of course, one dependency is the decomposition of footprints, however, the trace information to model information is also useful. For instance, the node "H/c" (depicted in tree form in Figure 3 and in graph form in Figure 5) traces to model elements [c/d] [2]. Since node "H/c" is also a child of node "B," it can be assumed that node "B" uses *all* model elements of node "H/c" (making again a consistency assumption; also recall Figure 4). This implies that node "B" should not only trace to model elements [g/h] but also to model elements [c/d][3]. This activity of propagating model information upwards on the footprint tree can be repeated for all child nodes. We refer to this activity as *Generalizing* since model information is generalized from child nodes to their parent nodes. By generalizing model information from the child "H/c" to the parent "B," we can make two trace assumptions:

1. Scenario "B" traces to the classes *Request* and *Inbox*
2. Classes *Request* and *Inbox* trace to data flow elements *Inbox* and *GetMail*

The first assumption reveals a trace from a scenario to model elements (trace type (c) in Figure 1) and the second assumption reveals a trace between two groups of model elements (trace type (d) in Figure 1). The latter assumption is still ambiguous since it implies that a subset of the two classes must trace to a subset of the two data flow elements. Nevertheless, previously, we did not have knowledge about traces between them, now we have some information.

Generalizing affects two additional nodes ("F" and "E") in Figure 5. After propagating model information from their children we find that the node "E" now traces to model elements [a/c/d/g/h] and that node "F" now traces to model elements [a/b/c/d/g/h]. This also implies that model element "a" (class *PopreadApp*) also traces to the data flow elements [g/h]. We already know that model elements [c/d] trace to model elements [g/h].

7 REFINING

Through *generalizing* a large number of new trace information becomes available. However, we still have a fair amount of unexplored nodes. For instance, in case of the node "AF" in Figure 5 it remains unknown as to what model elements it traces to. Knowing about trace informa-

[2] Note that we disregard the previous contradiction with respect to these elements for the sake of this exercise.

[3] Note that there is no new contradiction here since elements [c/d] and [g/h] are of different diagrams

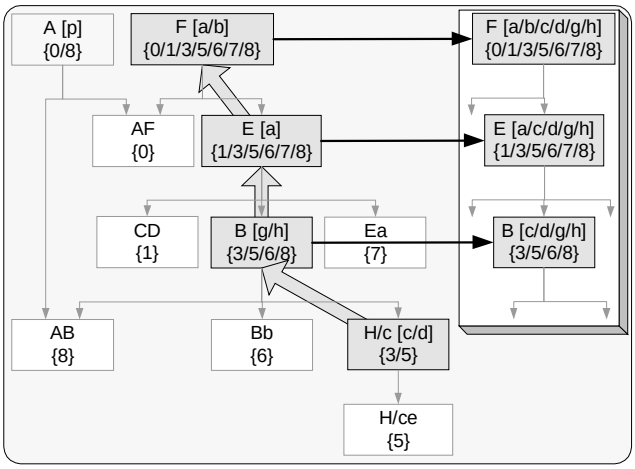

Figure 5. Generalizing model information

tion for node "AF" would be of great value since it could bridge the nodes "A" and "F." *Refining*, the final activity of our trace analyzer approach, takes information from parents and propagates it down to their children. As such, *refining* tries to induce what implications model elements on the parent's side might have onto their children. *Refining* is, however, not as simple as generalizing and depends on a number of conditions. Figure 6 depicts the refinement activity in context of the sub graph from Figure 5.

Take, for instance, node "A" that traces to model element [p] and also has two children called "AF" and "AB." Since its children cover the same footprints as their parent, it follows that they, together, must also trace to model element [p] (consistency assumption). Since node "A" refers to a single model element this implies that nodes "AF" and "AB" respectively must trace to a subset of this single model element. Since we do not split up model elements at this point, it follows that nodes "AF" and "AB" must both trace to [p].

A similar argument holds for node "F" and its children "AF" and "E." The difference to the case above is that one child ("E") already has substantial trace information available. Making again the consistency assumption, we can assume that node "AF" must refer to all those model elements of node "F" to which node "E" is not referring to. For instance, node "F" has a trace to model element [b] to which node "E" does not trace. Since nodes "AF" and "E" together also *completely* cover all footprints of node "F" it follows that node "AF" must trace to model element [b]. It is also possible that node "AF" traces to all model elements of node "E," however, this may not be true in all cases. In order to avoid incorrect model element propagation we have to ignore model elements [a/c/d/g/h]. After *refining*, node "AF" has two new traces implying that model element [b] (class *UserInterface*) must trace to model element [p] (use case *About*).

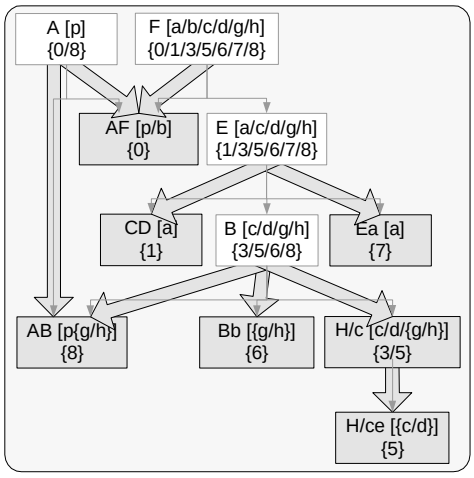

Figure 6. Refining model information

Node "E" and its children "CD," "B," and "Ea" is another example. Previously, we found node "E" to trace to model elements [a/c/d/g/h]. This, however, implies that node "E" now has a trace to model element [a] for which it has no child. Making the same assumption as above, we can infer that model element [a] can be propagated to the two child nodes "CD" and "Ea."

Node "B" describes a special case. Node "B" traces to two model elements of the class diagram [c/d] and two other model elements of the data flow diagram [g/h]. Like in some previous cases, traces to model elements of different diagrams have to be treated separately. Thus in this case, the model elements for the class diagram have to be investigated separately from the model elements for the data flow diagram. Since currently the child "H/c" already occupies both trace links to the class diagram we encounter a contradiction in that there are no class diagram elements left for nodes "AB" and "Bb." Simply speaking, if node "B" traces to two classes then each child of that node must trace to a non-empty subset of it (consistency assumption). We could assume scenario "Bb" and "AB" to also trace to [c/d], however, this assumption may not be valid in all cases.

Node "B" also makes references to the data flow elements [g/h]. In this case we find that none of its children refers to any data flow element. Again it is assumed that each child must refer to a non-empty subset of [g/h] resulting in ambiguous traces from "AB," "Bb," and "H/c" to [{g/h}]. The ambiguity is indicated in form of curly brackets within the model element list. In case of node "H/c" it is assumed that it traces to both "c," "d," and to either one or both of {g/h}." The refinement of "H/c" is another special case. Its child node "H/ce" has only a subset of the footprints of node "H/c." Thus, node "H/ce" may potentially only refer to a subset of the model elements of "H/c." It follows that the model elements [c/d] are propagated and declared ambiguous.

As can be seen in above cases, the activity of *refining* is more complex than *generalizing* and also results in numerous ambiguities. Nevertheless, *refining* also added new trace information and it paved the foundation for a subsequent generalization. For instance, the model information

in node "AF" could now be generalized again to its parents, resulting in additional trace information to other scenarios (e.g., model element "b" could be propagated to scenario "A"). The activities of *generalizing* and *refining* have to be repeated as long as model information is propagated.

Due to the limited space we are not able to elaborate on all propagation rules we have discovered. However, this paper summarized the most important situations.

8 AUTOMATION AND TOOL SUPPORT

Our tool, called TraceAnalyzer, supports our approach by automating the activities *atomizing*, *generalizing*, and *refining*. Figure 7 depicts a few screen snapshots of our tool. The lower left part of the figure depicts the main window showing the footprint graph and generalized trace information. Our tool takes observed trace information (e.g., as generated via Rational PureCoverage in lower-right of Figure 7) as well as hypothesized trace information (upper-left of Figure 7) as input. Our tool then generates the footprint graph and propagates trace information up and down that graph.

We currently do not employ automated testing tools. Instead, the testing of scenarios has to be done manually. Future works are the integration of our tool with a testing and a modeling environment since they together could eliminate the need for users to directly interact with our tool. Instead, trace information would be supplied by the testing tool and hypothesized trace information would be supplied by the modeling tool. Our tool could then perform trace analyses in the background, continuously updating the modeling tool with new trace information. We currently have extensive experience with working with modeling tools like Rational Rose. We previously developed an automated consistency-checking tool called UML/Analyzer [4] that would greatly benefit from such integration.

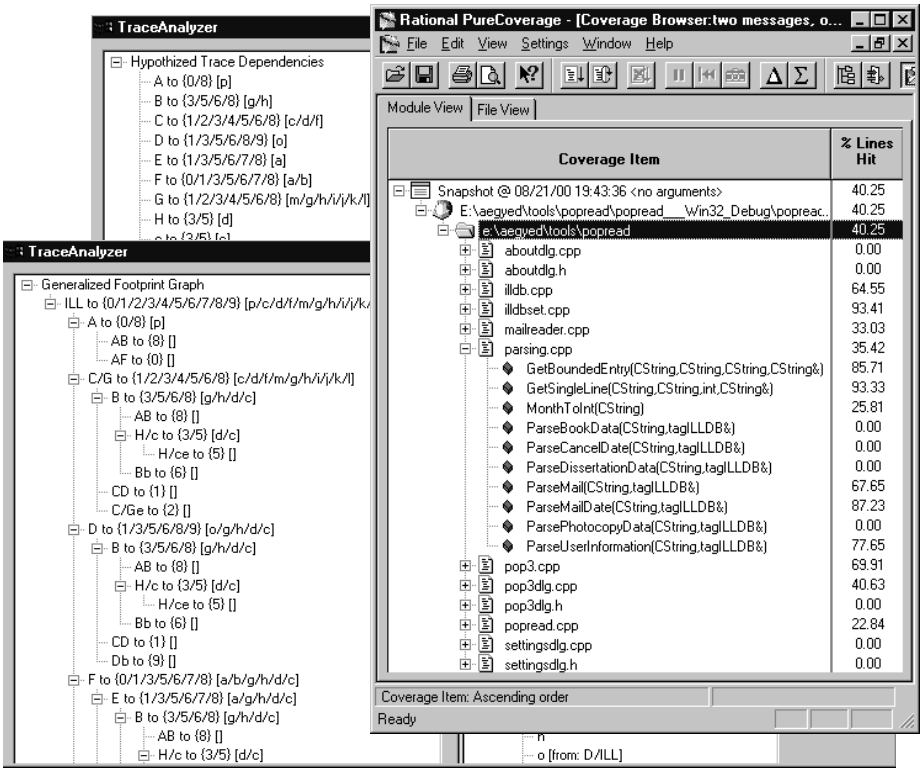

Figure 7. Trace Analyzer tool to support scenario-based trace analysis

9 DISCUSSION

Quality of Trace Generation and Validation

In this paper, we presented an approach for observing, generating, and validating trace information. We started out with only a few hypotheses, however, in the course of analyzing the given information, our approach was able to generate new trace information and invalidate existing ones. The following gives a short list of previously unknown traces:

Traces between model elements and system:
- Class *PopreadApp* traces to *CApp*
- Use case *About* traces to *CMainWin*

Traces between scenarios and model elements:
- Scenario "H" traces to model element *Request*
- Scenario "A" traces to model element *UserInterface*

Traces between model elements:
- Use case *About* traces to class *UserInterface*
- Class *PopreadApp* traces to class *Request*

Upon inspecting the quality of the newly generated traces, we indeed find that they fit the problem well. However, we also detected a series of conflicts that may indicate that some of the derived traces are incorrect. For instance, we learned that, while *refining* model elements [c/d] from node "E" to its children, a conflict was found in that one child (node "H/c") already traces to all classes its parent node is tracing to. This case, like others, pinpoint problems where either to little information was given or an inconsistency exists. A user needs to manually investigate these special cases to resolve the conflict. In above case, it turns out that the given hypothesized trace from scenario "B" to model elements [c/d] is incomplete.

Our approach generates ambiguous traces and contradictions if insufficient or incorrect trace information is provided. In order to improve the quality of the results, two measures can be taken: (1) the ambiguity of trace information decreases the more hypothesized trace information are available, and (2) the number of contradictions decrease with the degree of correctness of hypothesized traces.

Analysis Granularity

In this example we used implemen-

tation classes as the most atomic footprints. Implementation classes can, however, be very complex and may not always be the best choice. Upon validating our technique on other projects, we found that footprints should be generally analyzed on the level of class methods. While inspecting the ILL system, we already encountered a reason why finer granularity results in better quality traces. In case of node "AB" we found that it must trace to use case *About* and to either the data flow elements *Inbox* or *GetMail*. These traces are, however, counter-intuitive since displaying the *About* box should have nothing in common with checking for mail. The reason why our approach generated a connection between them was because they both use the same class *CMainWin*. Upon closer inspection of that class we, however, find that displaying the about box uses its method *OnAbout()* whereas checking for mail uses its method *OnCheckRequest()* (two distinct parts of the *CMainWin* class). If we were to repeat our analysis on a finer level of granularity (e.g., methods) then our approach would not detect a relationship between the use case *About* and the data flow elements *Inbox* and *GetMail*. In this paper, however, the example would have become too bulky to be of any use for demonstration purposes. It must be noted that both our approach and our tool support finer levels of granularity.

Legacy Systems and COTS components

The ILL is a third-party software system and we were not involved in its development. Furthermore, we did not have the benefit of interacting with its developers while performing this study. Our situation is therefore comparable to the situation of any development team that is asked to reuse old legacy systems. While analyzing the ILL system we had to use a number of trial-and-error steps to find hypothesized traces that result in only few contradictions. Our approach supports this type of explorative form of trace detection since "success" can be measured in the number of ambiguities and conflicts produced.

Our example also used COTS components. For one, the Microsoft Foundation Classes® (MFC) and a socket library were used to display information and to access the POP3 server. Furthermore, Microsoft Access® was used to store book requests on a back-end database. Although, we were not able to observe the internal workings of those components, our approach nevertheless was able to generate trace information to them since observable wrappers covered those COTS packages. Those wrappers thus became a substitute for the COTS packages.

Functional and Object-Oriented Decomposition

An interesting feature of the chosen example is that it was implemented in C++ but mixed functional and object oriented decomposition. As such the *parsing* component was implemented in a functional manner whereas the rest of the system was implement in an object-oriented manner. To complicate matters, the model also mixed functional and object-oriented styles (data flow diagram versus class diagram). The fact that our approach uses the source code as a foundation ignores conceptual boundaries. This property also overlaps with some of the advanced separation of concerns work (e.g., [16]) making our approach potentially very useful in that domain as well.

10 RELATED WORKS

Current literature contains ample publications about the need for traceability [6,17], however, when it comes to how to generate or validate trace information, the list becomes very short.

The works of Gotel and Finkelstein [6] talk about the traceability problem and why it still exists. One reason they believe to be the main cause is the lack of pre-requirements traceability. They argue that tracing requirements, as many only do, is not sufficient since requirements do not capture rationale. We agree with their point, however, we also believe another reason to be the lack of traceability throughout the entire development project, including the design. Our works thus also enables design traceability.

Lange and Nakamura present a program for tracing and visualization [10]. They took a similar approach as we did since they observe run-time behaviors of software systems. They also capture trace information, however, their emphasis is on how to visualize trace information and they do not compare them to generate or validate trace information.

Pinheiro and Goguen [12] took a very formal approach to requirements traceability. Their approach, called TOOR, addresses traceability by reasoning about technical and social factors. To that end, they devise an elaborate network of trace dependencies and transitive rules among them. Their approach is however, mostly useful in context requirements tracing, ignoring the problem of design (diagram) traceability. Murphy et al. [11] present a different but formal technique where source code information is abstracted to match higher-level model elements. They, however, use their abstractions for consistency checking between model and code but less so to infer trace information between different model elements as we do.

Concept analysis is a technique similar to our work on *atomizing*. Concept analysis, i.e., as used for reengineering of class hierarchies [14], provides a structured way of grouping binary dependencies. These groupings can then be formed into a concept lattice that is similar in nature to our footprint graph. It is unclear, however, whether concept analysis can be used to group three-dimensional artifacts as required in the footprint graph.

The approaches of Haumer et al. [7] and Jackson [9] constitute a small sample of manual traceability technique. Some of them try to infer traces based on keywords whereas others try to use a rich set of media (e.g., video, audio, etc.) to capture and maintain "trace rationale." Their works, however, only provide processes but no automated support (except for capturing traces). As our example has shown, traceability of even a small system can get very complex. Manual traceability detection, though effective, can thus become very costly.

Despite some deficiencies of above approaches, all techniques discussed above are useful since they cover traceability issues outside our domain or could be used to derive initial hypothesized trace information needed for our approach.

11 CONCLUSION

We belief that traces are the "blood vessels" of models (e.g., diagrams). These blood vessels act as their lifelines and raise mere fanciful depictions of boxes and arrows into legitimate, up-to-date, and useful representations of software systems. With the absence of trace information or with the uncertainty of their correctness, the usefulness of models is severely limited. To date, however, creating and validating trace information has been proven to be extremely time consuming and error-prone, resulting in high cost. This is one of the reasons it is not often (comprehensively) done.

In this paper we introduced a approach for detecting new trace information and for validating old ones. Our approach is automated and tool supported. With our approach we follow the belief that trace information has to be generated and validated iteratively; contrary the old practice of "develop first, document later." Our approach can and should be used in a highly iterative manner where previously detected traces become the future hypothesized traces. Its ability to "feed" onto previous results enables an incremental approach to traceability generation and validation. This incremental aspect makes our approach also very suitable towards forward engineering since trace analyses of sub systems (or sub components, i.e., legacy systems) can be used as input to validate the system itself (e.g., validate the parts then validate the whole). Thus, the hierarchical decomposition of software systems into smaller sub components and their prior individual testing provides executable software systems early on. Our approach is naturally also applicable to reverse engineering of recently development systems or legacy systems.

Future works are the integration of our approach with existing automated testing tools as well as modeling tools. It is foreseeable that such an integration could greatly reduce the already little manual labor in using our approach. Instead whenever a new (sub) system is tested, a testing tool could automatically trigger a new trace validation and the required hypothesized trace information could be extracted from a modeling tool (e.g., UML incorporates trace types in its meta language).

ACKNOWLEDGEMENTS

The author would like to thank Dave Wile and all anonymous reviewers for constructive feedback. Teknowledge Corporation did not support this work.

REFERENCES

1. Abi-Antoun, M., Ho, J., and Kwan, J. "Inter-Library Loan Management System: Revised Life-Cycle Architecture," USC-CSE, University of Southern California, Los Angeles, CA 90089-0781, USA.
2. Antoniol, G., Canfora, G., and De Lucia, A.: "Maintaining Traceability During Object-Oriented Software Evolution: A Case Study," *Proceedings of the IEEE International Conference on Software Maintenance*, 1998.
3. Clarke, S., Harrison, W., Ossher, H., and Tarr, P.: "Subject-Oriented Design: Towards Improved Alignment of Requirments, Design, and Code," *Proceedings of the 1999 ACM SIGPLAN Conferencer on Object-Oriented Programming, Systems, Languages, and Applications*, Dallas, TX, October 1998, pp.325-339.
4. Egyed, A.: "Heterogeneous View Integration and its Automation," PhD Dissertation, University of Southern California, Los Angeles, CA, August 2000.
5. Gieszl, L. R.: "Traceability for Integration," *Proceedings of the 2nd Conference on Systems Integration (ICSI 92)*, 1992, pp.220-228.
6. Gotel, O. C. Z. and Finkelstein, A. C. W.: "An Analysis of the Requirements Traceability Problem," *Proceedings of the First International Conference on Requirements Engineering*, 1994, pp.94-101.
7. Haumer, P., Pohl, K., Weidenhaupt, K., and Jarke, M.: "Improving Reviews by Extending Traceability," *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS)*, 1999.
8. Hughes, T. and Martin, C.: "Design Traceability of Complex Systems," *Proceedings of the 4th Annual Symposium on Human Interaction with Complex Systems*, 1998, pp.37-41.
9. Jackson, J.: "A Keyphrase Based Traceability Scheme," *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, 1991, pp.2-1-2/4.
10. Lange D. B. and Nakamura Y.: Object-Oriented Program Tracing and Visualization. *IEEE Computer* 30(5), 1997, 63-70.
11. Murphy, G. C., Notkin, D., and Sullivan, K.: "Software Reflexion Models: Bridging the Gap Between Source and High-Level Models," *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, October 1995, pp.18-28.
12. Pinheiro F. A. C. and Goguen J. A.: An Object-Oriented Tool for Tracing Requirements. *IEEE Software* 13(2), 1996, 52-64.
13. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.
14. Snelting, G. and Tip, F.: "Reengineering Class Hierachies Using Concept Analysis," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 1998, pp.99-110.
15. Swartout W. and Balzer R.: On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM* 25(7), 1982, 438-440.
16. Tarr, P., Osher, H., Harrison, W., and Sutton, S. M. Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering (ICSE 21)*, May 1999, pp.107-119.
17. Watkins R. and Neal M.: Why and How of Requirements Tracing. *IEEE Software* 11(4), 1994, 104-106.