Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), Irvine, CA, November 2004, pp. 3-12

Resolving Uncertainties during Trace Analysis

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
+1 310 578 5350 ext. 201
aegyed@acm.org

ABSTRACT

Software models provide independent perspectives onto software systems. Ideally, all models should use the same model element to describe the same part of a system. Practically, models elements are not shared because of syntactic and semantic differences among modeling notations. Trace dependencies explicitly maintain the commonalities among the distinct model elements.

Generating and maintaining trace dependencies is difficult, costly, and highly error-prone. Automated trace analysis techniques are scarce. This paper extends an existing, testing-based technique for generating and maintaining trace dependencies. It is based on the commonality principle: if two model elements of different perspectives are the same then they must have the same source code. The existing approach associates test scenarios with model elements, tests them, and observes what lines of code are being executed. Model elements are considered the same/similar if their testing uses the same/overlapping lines of code.

This paper extends the existing approach (and tool) by giving the user a richer, more powerful, yet precise language on how to relate model elements, test scenarios, and source code (the input). This allows some forms of uncertainties to exist in input data without sacrificing reliability. The extended approach also identifies "shared code." Shared code works against the commonality principle in that model elements do not relate if they overlap solely on their use of generic source code (e.g., queue). As a pre-requisite, our approach requires an executable and observable software system and test scenarios.

Categories and Subject Descriptors

D.2.10 [**Design**]: Trace Dependencies

General Terms

Documentation, Design.

1. INTRODUCTION

Separation of concerns [13] and aspect-oriented programming [11] are classical examples of cases where the complexity of modeling a system is divided up into perspectives. Within these perspectives, concerns are modeled and solved separately. Typically, specialized models are used. While development concerns may be modeled in separate perspectives, those concerns do affect one another. This raises the issue of consistency in that perspectives should not make contradictory assumptions. Synchronization mechanisms are required to ensure that all perspectives treat the system consistently.

This paper discusses a trace analysis technique for identifying trace dependencies among those model elements that represent same/similar aspects of a software system but are maintained separately. While finding trace dependencies alone is not sufficient to reconcile multiple perspectives, they are the foundation for any such mechanism. Understanding model dependencies is vital for guiding consistency checking, change propagation, reverse engineering, code generation, and many other activities. Their absence inhibits automation.

Since trace dependencies identify same/similar, albeit separately recorded development artifacts, knowledge about them is required at about any development stage: requirements engineering, design, implementation, testing, and maintenance.

Challenges to Overcome

Four key difficulties complicate trace analysis:

1) Complexity is n^2 for n model elements because every model element has a potential relationship to every other one.
2) Semantic and syntactic differences among models and their many-to-many mappings.
3) Unawareness on part of developer (different developers may solve different concerns).
4) Development information is captured informally (if at all), models are highly incomplete, they are based on a wide range of non-standard notations (or standardized notations are taken out of context), and they include "corporate knowledge" (e.g., terminology that has meaning to few people only).

Given all these difficulties, it is not surprising that developers have difficultly in creating trace dependencies and, once they are created, in maintaining them while the software system evolves [3]. This problem is known as the *traceability problem* [8].

To date, models rarely have explicit mechanisms for identifying how their model elements overlap with other models. Thus, identifying model dependencies is a predominantly manual and error-prone activity. Existing approaches for detecting trace dependencies require extensive manual intervention.

SIGSOFT'04/FSE-12, Oct. 31-Nov. 6, 2004, Newport Beach, CA, USA.

Necessary Ambiguities in Guidance

This paper discusses a technique for identifying trace dependencies that addresses all key difficulties discussed above. Our approach, originally introduced in [6], uses the executing software system and its testing as a baseline for finding trace dependencies. We demonstrated its usefulness on several case studies. Since *the technique requires the existence of some source code (any testable, partial implementation is sufficient), its use is restricted to the latter stages of the software lifecycle: implementation, testing, and maintenance.* This is not a severe restriction because these latter stages consume the bulk of software development cost and time [2]. These later stages also have the most need for trace dependencies while developers change and new requirements emerge. While the technique does not readily apply to requirements engineering and design, it is capable of generating trace dependencies for these artifacts later once an executable and testable software system is available.

While our approach solved some of the key difficulties of trace analysis, it required precise input and it made assumptions that did not hold always. This paper extends our approach to testing-based trace analysis to address these two issues. It allows developers to express uncertainties in input but it guarantees results to be precise and correct. It also addresses the problem of shared code that blurs the boundary of model elements. Giving developers the ability to express uncertainties reduces a major source of errors but it does not necessarily eliminate errors. This paper thus also demonstrates how to detect errors.

2. TRACE ANALYSIS (BACKGROUND)

The Trace Analyzer [6] (the approach and its tool) uses testing to generate trace dependencies. Testing a system results in lines of code being executed. This is observed automatically with off-the-shelf monitoring tools (e.g., IBM Rational Quantify). Since trace analysis aims at identifying same/similar model elements across multiple perspectives, it is expected that these same/similar model elements execute same/similar lines of code during testing. Therefore, if two tests (A and B) execute the same lines of code then there is a trace dependency. Otherwise, if two tests execute different lines of code then there cannot be a trace dependency.

The approach requires developers to associate tests with model elements (e.g., test A is about some model element X and test B is about some model element Y). If tests A and B execute same/similar lines of code then their model elements X and Y must be same/similar also. Thus, overlaps among lines of code imply trace dependencies through the commonality principle:

Commonality: if A is known to trace to some source code C_A and B is known to trace to some source code C_B then a trace dependency exists if C_A and C_B overlap

As input, the approach requires (1) model elements with unknown trace dependencies; (2) test scenarios that describe independent system tests; and (3) hypotheses on the how model elements relate to the test scenarios. By monitoring the lines of code executed by test cases, overlaps are identified. These overlaps imply trace dependencies among the test scenarios and subsequently among the model elements that are hypothesized to relate to those scenarios. Our approach is an improvement in that:

1) Only n input hypotheses are required to infer n^2 trace dependencies: a model element has trace dependencies with potentially every other model element (n^2) but a model element has one trace dependency to the system (n).
2) Semantic and syntactic differences among models are irrelevant: it is not required to understand the differences between any two models to determine trace dependencies (n^2 differences). Instead, it is only required to understand the difference in meaning between a model element, its test case, and the system (n differences).
3) Collaboration among developers is reduced: developers only need to investigate their own model elements and how they relate to the source code. There is no need to understanding any other developer's model.
4) The use of informal, partial, non-standardized notations is not a problem because these differences do not have to be understood in context of other models 2) or other developers 3).

The key benefit of our approach is the separation of concerns during trace analysis. Instead of having to understand the relationships among multiple perspectives, our approach only requires to understand the individual relationship between any perspective and the system. This relationship can be investigated fully independent of every other perspective.

The downside of our approach is that it detects trace dependencies among model elements only if they can be mapped to some source code. We refer to a model element's source code as its *footprint*. If a model element does not have a footprint then it cannot be tested. Therefore, this paper is only applicable to *product models* that describe software systems. This includes requirements models, design models (UML [15]), and architecture models but excludes process models or decision models. In the following, we will refer to product models simply as models.

Trace analysis is trivial if we know the precise footprint for every model element. Finding trace dependencies is then solely about applying the commonality principle. The most significant challenge of trace analysis is to determine the exact footprint for every model element. This requires transitive reasoning:

$$(m_1 \longrightarrow s_1) \wedge (s_1 \longrightarrow f_1) \Rightarrow (m_1 \longrightarrow f_1)$$

In [6], we assumed that a model element m has some footprint f if the model element has test scenarios s and these test scenarios have a combined footprint f. Unfortunately, there is no guarantee that developers correctly associate test scenarios with model elements ($m_1 \longrightarrow s_1$) (note that $s_1 \longrightarrow f_1$ is assumed to be correct because it can be observed automatically during testing). This may lead to two problems.

First, if test scenarios are not exhaustive in evaluating the scope of a model element then not all lines of code are being executed. This leads to a subset of the true footprint:

$$(m_1 \longrightarrow s_1) \wedge (s_1 \longrightarrow f_1) \Rightarrow (m_1 \longrightarrow f_x) \; where \; (f_1 \subseteq f_x)$$

Second, if developers are not aware of all model elements used by a test scenario then the given model elements appear to have a larger footprint than the true one:

$$(m_1 \longrightarrow s_1) \wedge (s_1 \longrightarrow f_1) \Rightarrow (m_x \longrightarrow f_1) \; where \; (m_1 \subseteq m_x)$$

This is a dilemma because every input has unknown uncertainties with contradictory effects. Our original trace analyzer was not able to handle these two problems. Furthermore, our approach was not able to identify shared code that is executed by multiple model elements but does not belong to any of them (i.e., shared code contains domain-independent knowledge).

3. PROBLEM STATEMENT

Using a system as a base line reduces trace analysis to two steps: (1) identifying the footprint of every model element and (2) identifying trace dependencies through overlaps among footprints.

Handling Uncertainties

If a model element *m is exactly* some footprint *f* then we know that *m* traces to the footprint and none other; and that the footprint belongs to *m* and none other. This is precise and complete input. Unfortunately, developers often have an incomplete and a potentially inconsistent understanding of how model elements relate to source code. We refer to this as *uncertainty*.

We speak of a *partiality uncertainty* if it is not known fully what something is. For example, a developer may be uncertain whether sufficient tests were performed to establish the footprint for a model element. Such an uncertainty is expressed by defining the model element to be *at least* the given footprint. If developers are uncertain whether the given model elements capture a test scenario completely then this uncertainty is addressed by stating that the model element is *at most* the given footprint. At times, developers may find it useful to say what a model element *is not*.

We speak of a *cluster uncertainty* if the role of individual elements within a group of elements is unknown. For example, developers may be uncertain about the role of individual lines of code but they may understand well the purpose of a collection of lines of code (e.g., methods, classes, packages). This applies to model elements also. Such uncertainties are expressed by clustering individual model elements. For instance, it may be easier to state that model elements 'a' and 'b' together relate to footprint '1' and '2' while it remains uncertain whether model element 'a' relates to '1,' '2,' or both.

Uncertainties give designers a wide repertoire for defining input relationships in a detail they feel comfortable with. Uncertainties are vital to developers who do not know everything. In our experience, not knowing everything is the norm and not the exception. Unfortunately, uncertainties obscure our knowledge on the precise and complete footprint of model elements. This paper will discuss how to identify the footprint for model elements in spite of uncertainties.

Handling Shared Code

We define shared code as application-independent source code that is used by more than one model element. Typically, shared code is general-purpose code (stack, queue), libraries (math, file IO, user interface), or other application-independent functionality. While shared code is used by multiple model elements, it should not be considered part of any model element. Unfortunately, shared code is problematic for testing-based trace analysis because it is being executed during the testing of any model element that uses the shared code. This confuses our approach into believing that shared code is overlapping footprint among model elements and it will generate trace dependencies.

Any two model elements should not be considered same/similar simply because they use the same shared code. Thus, if any two model elements overlap solely in their use of shared code then this should not lead to a trace dependency. The commonality principle does not apply to shared code. This paper will also discuss how to identify and isolate shared code.

4. TRACE ANALYSIS REVISITED

The following will introduce an extended technique for trace analysis to handle uncertainties and shared code.

4.1 Goal

Trace analysis has two goals. The first goal is to identify for every line of code the model element(s) it belongs to. If a line of code belongs to a model element then we say that the model element is *included*. If a line of code does not belong to model element then we say that it is *excluded*. If it is unknown whether a line of code belongs to a model element then the model element is neither included nor excluded. The second goal of trace analysis is to identify all lines of code that are shared. Shared code must be ignored during trace analysis since it is excluded from the commonality principle.

Trace analysis *is complete* if every line of code either includes or excludes every model element; or if the line of code is shared. We refer to the complete set of model elements as *M* and to the complete set of lines of code as *F* (for footprint of model elements). The goal of trace analysis is summarized as:

$$\forall_{fe \in F} \left(included(fe) \cup excluded(fe) = M \right) \vee \left(isShared(fe) \right)$$

If the trace analysis is not complete then the generated trace dependencies have uncertainties. Additional input (guidance) is required to complete the trace analysis. Even if the trace analysis is incomplete, it will generate useful results as will be shown.

4.2 Input (Guidance)

To support the cluster uncertainty in input, developers may group model elements and lines of code. Input on how model elements relate to lines of code thus may be given on the basis of individual elements or sets. The input $m \longrightarrow f$ clearly defines *m* to trace to *f*. We refer to *f* being the footprint of *m* where *f* may be any subset of *F*. If *m* is a set of model elements (e.g., m_1 and m_2) and if *f* is a set of lines of code (e.g., f_1 and f_2) then all model elements in *m* together trace to *f*. In other words, every subset of *m* traces to a subset of *f* so that the union of all subsets is exactly *f*.

$$m \longrightarrow f \Rightarrow \left(\forall_{m' \subseteq m} m' \longrightarrow f' \wedge f' \subseteq f \right) \wedge \left(\forall_{fe \in f} included(fe) \subseteq m \right)$$

To support the partiality uncertainty, developers may qualify the input between model elements and lines of code as "is", "isAtMost", "isAtLeast", "isNot", "isExactly". Their exact implications are discussed later. We require input (guidance) to be correct to guarantee correct results but safeguards exist that may detect incorrect guidance.

If a source code (footprint) of an input overlaps with the source code of another input then the source code is fragmented into several parts. The overlapping source code is one fragment and the source codes for each of the non-overlapping parts are the other fragments. We will denote "code elements" to be the fragments of source code caused by the overlaps among all inputs. Input is then expressed in terms of code elements, which has the advantage that every input refers fully or not at all to all lines of code within a code element. In other words, all lines of code within code elements relate to model elements in the exact same way and need not be considered separately during trace analysis. This is a useful optimization to minimize the number of objects required to represent the system.

4.3 Examples

This section shows examples of various inputs to discuss the basics of trace analysis.

4.3.1 Simple Illustration

Consider the two illustrations in Figure 1. Figure 1 (a), first row, depicts the input that model elements *a* and *b* together relate *exactly* to code elements *1* and *2* (note: code elements *1* and *2* refer to distinct sets of lines of code). In other words, the model elements {*a*,*b*} have the footprint {*1*,*2*} exactly and none other.

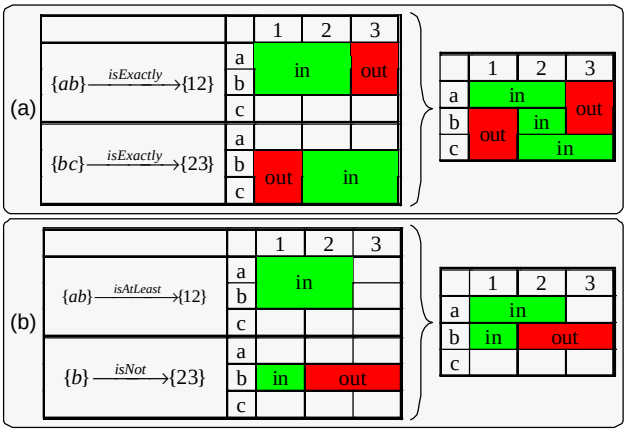

Figure 1. Combining Input Data (two illustrations)

Our approach annotates code elements with the properties of included/excluded lists to capture the input. Figure 1 illustrates these graphically using colored regions to highlight included (in) and excluded (out) model elements (rows) for code elements (columns). The input {*a*,*b*} is exactly {*1*,*2*} results in *a* and *b* being included in both *1* and 2 (medium gray/green region labeled "in") and excluded in *3* (dark gray/red region labeled "out").

Figure 1 (a) also depicts the second input that model elements {*b*,*c*} relate exactly to {*2*,*3*}. This leads to a different set of included and excluded model elements among the same code elements. Both inputs overlap in their use of the code element *2*.

The right part of Figure 1 (a) shows the combined input. There *b* is excluded from *3* and from *1* because the first input excluded it from *3* and the second input excluded it from *1*. The combined input also shows *b* to be included in *2* which is a logical consequence of its exclusion everywhere else. This reasoning is based on the assumption that every model element must own some unique aspect of the system. In other words, every model element must have some footprint (recall Section 2).

$$validModelElements(m) = \left(\forall_{me \in m} \exists_{fe \in f} included(fe) = me \right)$$

Since model element *b* must relate to a subset of {*1*,*2*} (first input) but it is not allowed to relate to *1*, it follows that *b* must relate to {1,2}-{1}={2}. Of course, a model element may never be included and excluded within the same code element.

$$\forall_{fe \in F} included(fe) \cap excluded(fe) = \{\}$$

Figure 1 (b) shows another illustration. The first input defines model elements {*a*,*b*} to relate to at least *1* and *2*. This implies that model elements *a* and *b* are not excluded anywhere because "at least" implies that {*a*,*b*} may relate to a footprint larger than {*1*,*2*} (e.g., 3). The second input defines *b not* to be *2* and *3*. The combined input again resolves some uncertainty using similar reasoning as above.

Thus, trace analysis is a straightforward form of analytical reasoning. That is, ignoring shared code and *perspectives*.

4.3.2 Illustration with Perspectives

Trace analysis identifies similarities among model elements that are captured separately in perspectives (recall Section 0). An example of a perspective is a class diagram (e.g. UML). Every class in a class diagram has a unique purpose since it describes a unique aspect of the system. It follows that every class must have some unique code not shared with any other class. Another example of a perspective is a state chart diagram where every state transition describes a unique behavioral aspect of the system.

If some form of behavioral description (e.g., statechart diagram) complements the structural description (e.g., class diagram) then there are two perspectives of the same system. Both perspectives use syntactically and semantically different model elements. However, both perspectives describe the same system. It follows that the structure has to accommodate the behavior and vice versa. Trace analysis reveals what part of the structure has to accommodate what part of the behavior.

Knowledge about perspectives is an optional, useful input to trace analysis. If a developer defines a set of model elements to form a perspective then every model element of the perspective represents something unique about the system:

$$unique(m) \Rightarrow \left(\forall_{me \in m} \exists_{fe \in f} included(fe) = me \wedge \neg isShared(fe) \right)$$

This implies that every model element within a perspective has some unique code elements not shared with any other model element. Therefore, the model elements of a perspective complement one another. Developers either identify perspectives manually or implicitly through the types of model elements used (i.e., classes in a class diagram always form a perspective).

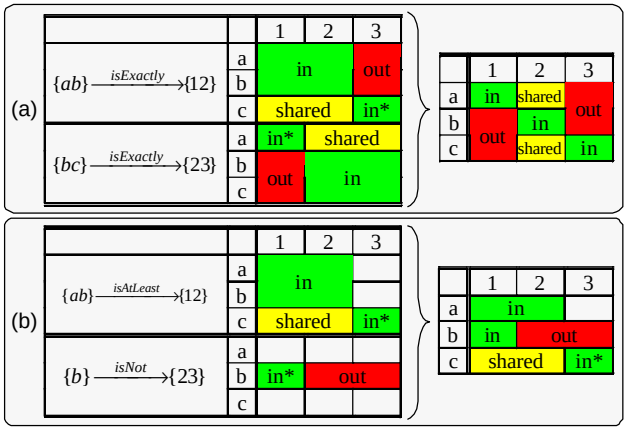

Figure 2. Combining Input Data with Perspectives

Figure 2 revisits the illustrations from Figure 1 with the difference that model elements *a*, *b*, and *c* are part of the same perspective. Again, Figure 1 (a), first row, shows that {*a*,*b*} is included in {*1*,*2*} and excluded in {*3*} because model elements {*a*,*b*} trace exactly to footprints {*1*,*2*}.

Since every model element in a perspective needs to contribute some unique aspect to the system, the input implies that *1* and *2* must be the unique part for *a* and *b*. Thus, model element *c* (the remaining model element of the same perspective as *a* and *b*) needs to find its unique code somewhere else. Only code element *3* remains and it follows that model element *c* must relate to a subset of *3*. Why a subset? A perspective describes a particular point of view onto a system. Its point of view may not cover the entire system but a subset only. Through the first input, we know that the footprint of the perspective is *at least* {*1,2*}. Given that we do not know whether the scope of the perspective is about the entire system, we must assume that the perspective is *at most* {*1,2,3*}. This uncertainty is captured in Figure 2 in form of a star symbol ("*") next to the label "in."

The input in Figure 1 (a), first row, also defines footprints of any model element other than *a* and *b* to be shared. This is another logical consequence of the uniqueness of model elements in perspectives. If the input defines model elements *a* and *b* to be footprint *1* and *2* exactly then a subset of that footprint must be unique to *a*, another subset must be unique to *b*, and the remaining subset, if any, must be shared code (note: these three subsets may not overlap => unique). While we cannot infer what these subsets are, we can assume that no other model element of the same perspective may find its unique code in {*1,2*}. Thus, if other input should state that another model element of the same perspective is included in any subset of {*1,2*} then this subset must be shared code (light gray/yellow region labeled "shared").

We thus use a third list, called *shared*, (in addition to *included* and *excluded*) to remember *potential* shared code. For example, with the input {*a,b*} is exactly {*1,2*}, we add the model element *c* to the *shared* list of both *1* and *2*. If, later, model element *c* is added to the list of included elements of either *1* or *2* then we know that this footprint is shared code:

$$isShared(fe) = included(fe) \cap shared(fe) \neq e$$

4.4 Footprint Graph

Any two inputs may overlap partially or fully in the footprints they cover and the model elements they use (they may overlap completely, partially, or not). Depending on the types of overlaps and the many combinations, there are over 400 scenarios for understanding the relationships between any two inputs. The previous section presented two illustrations only. It is thus impractical to present rules on how inputs affect one another. Instead, this section will introduce a generic structure, called the footprint graph (originally introduced in [6]), to discuss how input is translated and interpreted. We will discuss:

1) structure for capturing input (guidance)
2) translating input into structure
3) refining structure

4.4.1 Structure for Capturing Input

The footprint graph is organized around code elements because generating trace dependencies requires the understanding of overlaps among the code elements. The purpose of the footprint graph is to represent input in a uniform manner regardless of the input type so that their effects can be combined easily.

Nodes in the graph represent individual code elements or sets of code elements. Every node has the three lists *included*, *excluded*, and *shared* (see Figure 3). The model elements in the included list represent the part of the system covered by the node's footprint. The model elements in the excluded list do not represent any part of the node's footprint. The shared list contains a list of model elements that should not be included in the same node unless the node represents shared code. Initially, these lists are empty.

Figure 3 revisits the two illustrations introduced in Figure 2. It shows the nodes that are created for each illustration and the values they are populated with. We still assume that {*a,b,c*} belong to the same perspective. We learned from Figure 2 (a) and its discussion that the input "*ab* is exactly *12*" is equivalent to {*1,2*} including {*a,b*} and sharing *c*; and *3* excluding {*a,b*} and including *c*. Thus, we require two nodes. Node *12* included=*ab* and shared=*c*; node 3 excluded=*ab* and included=*c* (Figure 3 (a) depicts this and other information).

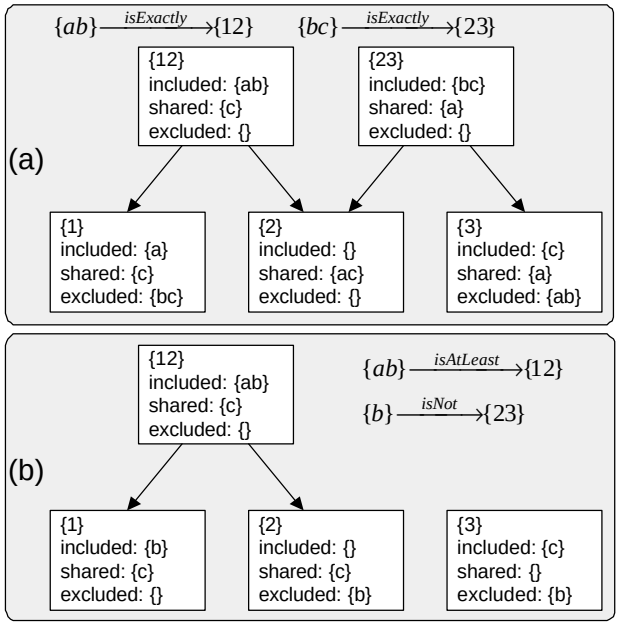

Figure 3. Footprint Graph for Two Sample Inputs

The footprint graph has edges to present overlaps among the code elements of nodes. An edge defines a parent-child relationship between two nodes where the child has a subset of the code elements of the parent. Nodes, connected through edges, form a hierarchy where the leaf nodes (bottom nodes) represent small pieces of code and the root nodes (top nodes) represent large parts of code or the entire system. This hierarchy is necessary because individual nodes fragment global knowledge, which leads to information loss. For example, node *12* including {*a,b*} is not equivalent to node *1* including *a* or *b* or both and node *2* including *a* or *b* or both. The latter two statements allow the possibility that both nodes are either *a* or *b* while the first statement requires one to be *a* and the other to be *b*.

Figure 3 (a) depicts "*ab* is exactly *12*" and "*bc* is exactly *23*" together. The first input requires nodes *12* and *3* while the second input requires nodes *23* and *1*. Edges exist because nodes 1 and 3 are subsets of nodes *12* and *23* respectively. Overlaps among nodes in the footprint graph are captured explicitly. Nodes *12* and *23* overlap in the code element *2* which results in the node *2* (recall the fragmentation discussed in Section 4.2).

However, Figure 3 only depicts the result of constructing the footprint graph. We will discuss later how overlaps and, consequently, their additional parent-child edges are used during trace analysis (e.g., to propagate "b" to node {2}).

The footprint graph has a series of properties. If a node includes a model element then every parent node must include the model element also. After all, the parent node refers to a superset of the code elements of its children. The parent thus "inherits" the union of included elements of all its children:

$$included(fe) = included(node(fe)) + \bigcup_{child \in children(node(fe))} included(child)$$

In reverse, the excluded elements of a child "inherits" the excluded ones from its parents. If some footprint excludes a model element then all subsets of that footprint must exclude it. Furthermore, the parent may "inherit" excluded model elements from its children if every child excludes those model elements and the footprint of the children together is equal the footprint of the parent. The same applies to shared elements. Neither case is computationally appealing but a useful observation simplifies matters.

For trace analysis, we are interested in knowing included, excluded, and shared elements for individual code elements only. Through the above observation, we know that every input that excludes/shares model elements for some footprint can be re-interpreted as the same elements excluding/sharing every code element individually. In the beginning, we thus add a node for every code element to the footprint graph. Since no node will ever have a footprint equal or less of these, we refer to these nodes as *footprint leaves*. Excluded/shared elements are added to every footprint leaf they overlap with. The excluded/shared elements of parents are then easily computed as:

$$excluded(f) = \bigcap_{leave \in leaves(f)} excluded(leave)$$

$$shared(f) = \bigcap_{leave \in leaves(f)} shared(leave)$$

For example, in Figure 3 (a), node *3* contains model element *a* in its shared list. This is because the statement "*23* sharing *a*" is re-interpreted as "*2* sharing *a*" and "*3* sharing *a*". Node *2* combines the shared elements from both its parents and, consequently, both *a* and *c* are part of the shared list of node *2*.

The footprint graph is minimal in that every node contains information that is not derivable automatically through its parents or children. For example, Figure 3 (b) depicts the footprint graph for the second illustration from Figure 2 (details of generating it are omitted given the similarity with the first illustration). What is missing from Figure 3 (b) is a node *23* that excludes *b*. It was discarded because its children, nodes *2* and *3* contain all knowledge of this parent.

4.4.2 Translating Input into Structure

To represent the input, we add nodes and edges. Depending on the type of input, one or two nodes are created, or they are updated if the nodes already exist.

The input $m \xrightarrow{is} f$ defines *m* to trace to *f*. Footprint *f* thus *includes* the model elements in *m*. This input contains uncertainties in that it does not define whether *m* traces to any footprint other than *f* (a subset of the remaining footprint *F-f*) or whether other model elements (a subset of the remaining model elements other than *m*) may trace to the same footprint. This uncertainty is

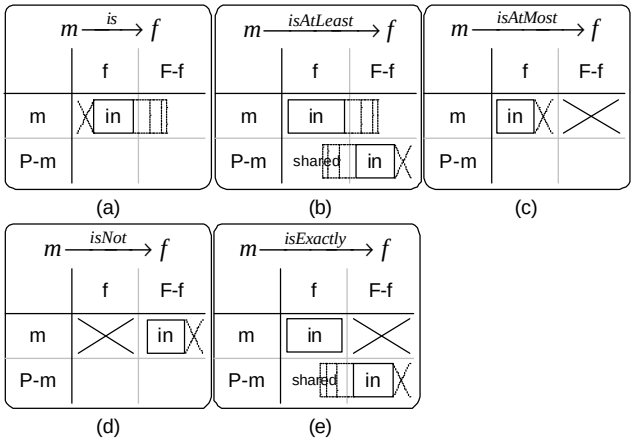

Figure 4. Basic Input Rules. More complex rules can be composed of these basic rules.

captured in the table in Figure 4 (a) as *m* being included in *f* (in the row *m* and column *f*) next to a dashed cross. The cross implies that not all of *f* may belong to *m*.

The input $m \xrightarrow{isAtLeast} f$ defines the minimal footprint of a set of model elements. The uncertainty in the statement "atLeast" implies that the true footprint of *m* is potentially larger than *f* (indicated through the box's dashed extension in Figure 4 (b)).

The situation is different if there are multiple model elements in the same perspective. We refer to these other model elements of the same perspective as *P-m*. We previously discussed that all model elements part of the same perspectives must have some unique code. Given that all of *f* belongs to *m*, it follows that all remaining model elements of the same perspective must relate to some code other than *f*. The footprint *F-f* thus includes the model elements *P-m*. And, it is possible to detect shared code if any subset of *f* includes any subset of *P-m* (recall Section 4.3.2).

The input $m \xrightarrow{isAtMost} f$ defines the maximal footprint of a set of model elements. The given model element must not have any footprint other than *f*. In other words, model elements *m* are excluded from footprint *F-f* and included in footprint *f* (see Figure 4 (c)). Also, "isAtMost" implies the uncertainty that *m* could have a footprint less than *f*. This uncertainty is expressed in the now familiar fashion of a dashed X. The "isAtMost" rule does not restrict in any way the model elements *P-m*. Clearly, it is possible that *P-m* is included in *f* but we have no evidence to support this. Likewise, it is possible that *P-m* is included in *F-f* but we have no evidence to support that either.

The input "isNot" is functionally the negation of "isAtMost". Given that every model element requires some footprint, it follows that if model elements in *m* are excluded from *f* then they must be included in *F-f* to have some footprint. As in the case of "isAtMost", the inclusion statement is accompanied by a dashed X implying that model elements other than *m* might also be included in *F-f*.

To summarize, this section demonstrated how to separate precise input from uncertain input. We found the presented five inputs the most useful ones in handling partiality and cluster uncertainty. Other inputs are possible but not discussed. The table below summarizes the effects of input with uncertainties onto nodes (i.e., it restates Figure 4 in terms of node properties). For example, the

input rule $m \xrightarrow{isAtLeast} f$ requires the two nodes *f* and *F-f*. Model elements in *m* are added to the included list of node *f*; model elements in *P-m* are added to the same node's shared list; and model elements in *P-m* are added to the excluded list of node *F-f*. The latter two additions have no effect if the model elements are not part of a perspective because *P-m* is empty (i.e., adding an empty set to an existing one does not change it).

Input	Nodes
$m \xrightarrow{is} f$	*included(f) += m*
$m \xrightarrow{isAtLeast} f$	*included(f) += m, shared(f) += P-m* *included(F-f) += P-m*
$m \xrightarrow{isAtMost} f$	*included(f) += m, excluded(F-f) += m*
$m \xrightarrow{isNot} f$	*included(F-f) += m, excluded(f) += m*
$m \xrightarrow{isExactly} f$	*included(f) += m, excluded(F-f) += m* *shared(f) += P-m, excluded(F-f) += P-m*

If a node does not exist then it is created with included, excluded, and shared already being empty. If a node with the same footprint exists then the node is updated by adding the new information. No information is ever deleted from a node. Edges are added between newly created nodes and their parents or children. We speak of a footprint graph because every child may have zero to many parents and every parent may have zero to many children.

4.4.3 Refining Structure

Once constructed, the footprint graph reflects the input and the "inherited" parent/child properties. The following discusses how to refine the structure by pushing included elements to children nodes. Refinement computes, with increasing detail, how individual code elements relate to model elements. Refinement furthers the goal of trace analysis expressed in Section 4.1.

$$excludedNodes(m) = \{c \in nodes \mid m \in excluded(c)\}$$

$$sharedNodes(m) = \{c \in children(f) \mid m \cap shared(c) \neq \{\}\}$$

$$\forall_{\substack{m \in included(f) \\ m \notin shared(f)}} included(f - excludedNodes(m) - sharedNodes(m)) = m$$

We know that every model element must have some footprint. If a node includes model element *m* and that node has some children that exclude or share model element *m* then the remaining footprint must include *m*. Figure 5 shows two applications of the refinement rule on the first illustration introduced in Figure 2.

The first application of the rule is in *b* getting refined from node *12* to node *2*. The reasoning is as follows: We know that *b* is included in *12* and that it is excluded from *1*. Since *b* must have a footprint, it must be in the remaining footprint *2*.

Similar reasoning refines *a* from node *12*. We know that *a* is included in *12* but shared in *2*. Thus, if *a* were included in *2*, this node would become shared code. Since *a* must have a unique, unshared footprint, it must be included in the remaining footprint *1*. Given that node *1* already included *a*, it does not add new knowledge in this case.

Both refinement scenarios move included model elements from parent nodes to children nodes. In case of nodes *1* and *3*, the nodes are complete now because they include and exclude all model elements of the same perspective; and they share nothing. Node *2* is still incomplete.

$$isComplete(fe) \Rightarrow P - (excluded(fe) \cup included(fe))$$

The star symbol ("*") annotates an incomplete list and it is a visual aid for the developer to identify incompleteness. The symbol indicates that other model elements of the same perspective may be added to the list. For example, node *2* is incomplete because it is possible that future input add model elements *a* or *c* to node *2* or a subset thereof.

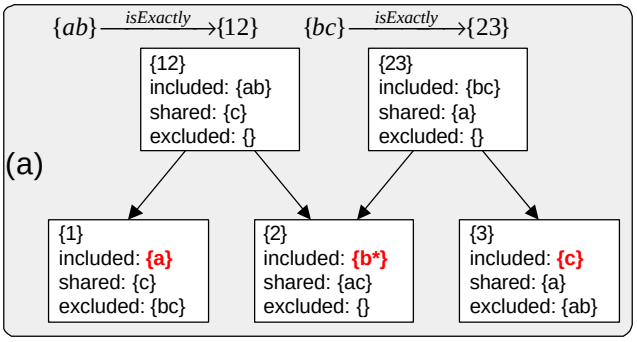

Figure 5. Refined Footprint Graph

In both refinement scenarios, a new node would have to be created if no such node would have existed already. Refinement starts at the root of the graph and works towards the leaves. If a model element is added to a node then the parents and children instantly "inherit" the newly added information. There is no race condition in what order model elements are refined (proof is excluded for brevity).

4.5 Generating Trace Dependenciess

Trace dependencies are derived from the nodes of the footprint graph. If two model elements overlap in the same node then there is a trace dependency between them. Given that a system may be fragmented into many small nodes, it is likely that the same model element is included in more than one node. In this case, the footprint of the model element is the union of all those nodes.

Since model elements of different perspectives are semantically and syntactically different, not many model elements will have the exact same footprint. In this case, the "similarity" between any two model elements is expressed in terms of how much source code (i.e., nodes) they have in common. The more common source code, the more similar they are.

The output of the trace analysis may contain uncertainties if some nodes remain incomplete. Developers may use knowledge of incompleteness to guide what additional input is required to make the trace analysis more complete. However, output with uncertainties is still useful. First, a developer may find it useful to have partially complete trace dependencies instead of having none. Second, the problem of incompleteness is reduced through grouping. Take, for example, the bottom of Figure 3 (b) before refinement. Node *1* includes model element *b* and node *2* is not (yet) known to trace to anything. If some other, third model element overlaps with nodes *1* and *2* then the result seems uncertain. After all, we do not know what is included in node *2*. Still, we know that {*1,2*} together include *a* and *b* because there is a parent node that says so. Thus, despite the uncertainty in both leaf nodes, it is possible to generate a trace dependency without uncertainty in this case. The details of how to generate trace dependencies is not discussed in more detail here (see [6]).

5. CASE STUDY AND TOOL SUPPORT

The following demonstrates the trace analysis on a video-on-demand system (VOD) [5]. The VOD plays selected movies on demand. The VOD was modeled in UML and Figure 6 depicts two UML diagrams (perspectives). The statechart diagram (top) describes the behavior of VOD. A user can select individual movies for playing. During playing, a selected movie may be paused, stopped, and played again. The transitions between these states correspond to buttons a user may press in the VOD's user interface. The class diagram (bottom) shows the coarse structural decomposition of VOD. In the following, the model elements are referred to by their short identifiers. Note that the presented model is a subset of the actual UML model for brevity.

The goal of the trace analysis is to understand how the statechart elements relate to the classes. There are 256 theoretical trace dependencies among the ten state transitions and six classes $(6+10)^2$. Every state transition describes a distinct behavior and every class describes a different part of VOD. Thus, they represent two separate perspectives of the VOD system. The goal of trace analysis is to identify the commonality between them. For example, what state transition requires the Streamer? Or what classes implement the "Play" transition? While it might be easy to guess some of those trace dependencies, the semi-formal nature of the UML diagrams makes it hard to identify complete and correct trace dependencies manually.

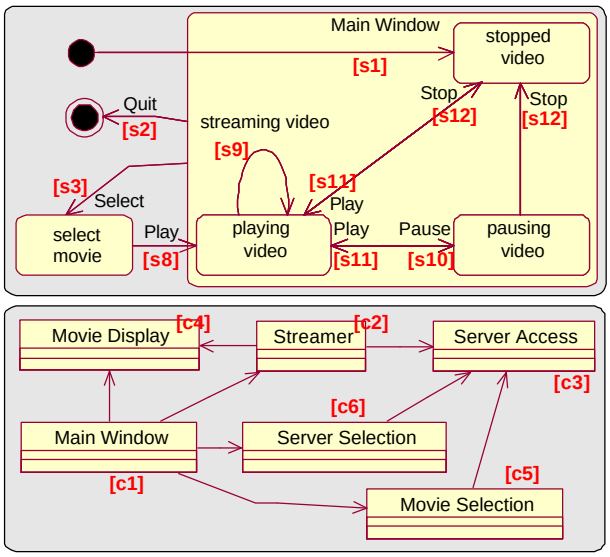

Figure 6. Video-On-Demand Case Study

The following shows how our trace analysis derives trace dependencies. The developer is required to define how model elements in individual perspectives relate to the 21 java classes (labeled A through U) and the 200+ methods. For brevity, we will assume that if any line of code of a java class relates to a model element then the whole java class relates to that model element. This is not correct for precise trace analysis but useful for demonstrative purposes since it limits our trace analysis to 21 code elements instead of thousands of them (see also Section 4.2).

Perspective: "class" cl {c1,c2,c3,c4,c5,c6}
#user interfaces
Dependency: {c5,c6} isAtLeast {C,J,S}
#playing a movie
Dependency: {c2,c3,c4} isExactly {A,C,D,F,G,I,K,O}
#finding and selecting a movie
Dependency: {c1,c3,c5} isAtMost {C,J,N,R,U}

The example input above defines the perspective for classes and it defines three input dependencies between classes and source code. For example, the first input dependency states that model elements c5 and c6 together are at least the java classes C, J, and S. Section 4.4.2 describes how to translate this input into the footprint graph. It requires the node {C,J,S} to include {c5,c6} and to share the remaining classes of the same perspective {c1,c2,c3,c4}. It also requires the remaining footprint {A,B,D-I,K-R,T,U} to include the remaining classes.

Figure 7 depicts the footprint graph for the entire input as it was generated by our Trace/Analyzer tool. The boxes and lines correspond to nodes and parent/child relationships. The top field in the box defines the footprint. The legend on the upper, right corner resolves how the footprint relates to java classes. The other three fields in the nodes define their included (in), excluded (ex), and shared (sh) model elements. The input example discussed above is reflected in the two nodes {1,3,4} and {2,5,6}.

In total, 11 nodes were created for the given three input rules (e.g., {1,3,4}). These nodes account for the input and overlaps among input nodes. Refinement then created the remaining three nodes (e.g., {1,2}, {1,4}) and modified the existing ones. The depicted footprint graph is not minimized to make the example easier to understand. The gray nodes can be eliminated safely because their children contain all information. Thus, the actual, minimized footprint graph is only eight nodes in size.

The three given input rules contain uncertainties in how model elements relate to java classes. Despite those uncertainties, refinement uncovers what individual model elements, or smaller groupings thereof, relate to individual footprint nodes (the goal of trace analysis is to find how model elements relate to lines of code). For example, through node {1} we learn that model element {c5} is related to java class J because node {1,4} includes {c5,c6} and one of its children, node {4} excludes {c5}. How do we know that {c5} is excluded in node {4}? The input that model elements {c1,c3,c5} isAtMost C, J, N, R, and U implies that {c5} must be within this footprint and that it is excluded from every other code elements (including java class S that is represented by node {4}).

The footprint graph also contains an inconsistency because node {2,5,6} includes and excludes model element {c3}. This is caused by a rather complex interplay among all three input dependencies. The input {c2,c3,c4} isExactly {A,C,D,F,G,I,K,O} defines model element {c3} to be within the given footprint while the input {c1,c3,c5} isAtMost {C,J,N,R,U} defines it to be within the other footprint. Upon closer inspection, this is not a conflict because java class C is within both footprints and thus would provide a unique footprint for {c3}. However, the input {c5,c6} isAtLeast {C,J,S} defines java class C to be about either c5 or c6. Consequently, there is a conflict in that the first two input dependencies require its exclusion from a given footprint while the last input dependency requires its inclusion in the same footprint. Obviously, this inconsistency is very hard to detect manually but our tool identifies it automatically. The source of the conflict is the dual role of the *Server Access* class {c3}. It provides services for streaming movies and for movie searching. However, the implementation separates this functionality into the two distinct java

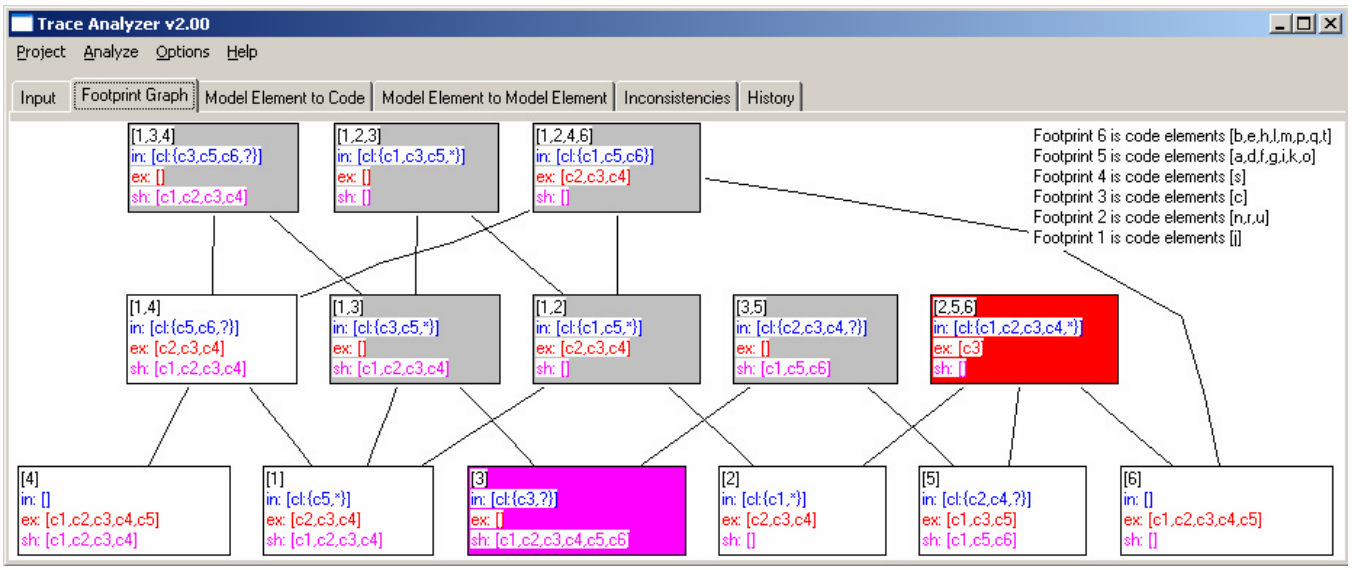

Figure 7. Trace/Analyzer Tool shows Refined (but not minimized) Footprint Graph for Video-On-Demand Case Study

classes A and R. Therefore, the developer was overconfident in using the isExactly and isAtMost declaration. The appropriate input would have been isAtLeast in both cases.

The footprint graph also contains examples of shared code. Footprint {3} represents the java class C which is used by all three input dependencies. Given that the first input {c5,c6} isAtLeast {C,J,S} and the second input {c2,c3,c4} isExactly {A,C,D,F,G,I,K,O} claim C to belong to distinct sets of model elements, it appears as if java class C is a general purpose support class that provides generic services. This is indeed true because the class provides a basic GUI canvas used by all displays.

This example demonstrates how uncertainties in input are resolved during trace analysis. However, this is only the first part of trace analysis. The second part is to identify trace dependencies among model elements of different perspectives. In order to identify trace dependencies between the classes and the state transitions, we also need to understand how the state transitions relate to the source code (java classes). For brevity, we introduce a single input for state transitions only.

#play, pause, stop movie
Dependency: {s09,s10,s11,s12} is {A,C,D,F,G,I,K,N,O,R,T,U}

This input rule defines that the play, pause, and stop state transitions use the indicated java classes during execution. If we compare this knowledge with the footprint graph then we find that it overlaps with nodes {2}, {3}, {5}, and {6}. Since these four nodes include model elements {c1,c2,c3,c4} we conclude that state transitions {s09,s10,s11,s12} have a trace dependency to {c1,c2,c3,c4} (note: to guarantee correctness we actually would have to resolve the inconsistency first).

6. VALIDATION

The trace analyzer technique and its tool support were evaluated on several case studies to date (e.g., [1,5]). This validation included a wide range of development artifacts (e.g., requirements [7], functional design languages such as data flow diagrams, object oriented design languages such as class diagrams, behaviors descriptions such as statechart diagrams), and different kinds of source code (e.g., C++, Java, Visual Basic).

We observed that the size of the footprint graph does not increase linearly with the number of input dependencies. While every input adds at most two new nodes (linear), additional nodes are created to capture their overlaps with existing nodes in the graph. Theoretically, the footprint graph may include a node for every combination of code elements (n^2). Practically, the sizes of the footprint graphs were much smaller. We contribute this to two optimizations: (1) we combine individual lines of code into larger code elements and (2) we add nodes only if they contain more information than their children do. The latter optimization, in particular, guarantees that the footprint graph is minimal. In fact, no parent nodes exist if the leaf nodes are complete (see *isComplete*). We have not yet performed exhaustive evaluations to measure the growth of the graph but the n^2 worst-case growth is reasonable given the previously discussed n^2 complexity of identifying trace dependencies.

We also evaluated the correctness of the trace analysis. We determined two causes of errors: (1) incorrect input (was discussed previously) and (2) overly granular code elements. The latter may not be obvious. Combining lines of code into code elements increases the likelihood that source code is combined that serves different purposes. During trace analysis, it then appears as if those different purposes are related. Our technique detects inconsistencies but their absence does not guarantee correctness. However, the more complete the trace analysis, the less likely the inconsistencies.

Shared code is problematic during trace analysis because it does not belong to any individual model element. This obstructs the finding of commonalities, which is obviously based on overlapping footprints. We demonstrated that we can detect the existence of shared code but this capability is limited.

7. RELATED WORK

Pinheiro and Goguen [14] devised an elaborate network of trace dependencies and transitive rules among them to support requirements traceability. Their approach, called TOOR, addresses traceability by reasoning about technical and social factors. Their approach is limited to requirements and ignores the problem of

traceability among development artifacts in general. Their work also ignores the problem of how to generate and validate trace dependencies among development artifacts that are not defined formally and completely.

Concept analysis (i.e., as used for the reengineering of class hierarchies [16]), 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 and interpret three-dimensional artifacts (code, scenarios, and model elements) as required in the footprint graph.

The approaches of Haumer et al. [9], Jackson [10], and Cox-Delugach [4] constitute a small sample of manual traceability techniques. Some of them infer traces based on keywords whereas others use a rich set of media (e.g., video, audio, etc.) to capture and maintain trace rationale. Their works only provide manual processes and do not automate trace generation and validation (except for capturing traces). As our example has shown, trace generation for even a small system can become very complex. Manual trace detection, though effective, can thus become very costly.

Our approach observes the source code of a software system according to test scenarios executed on it. This activity is similar to slicing where the source code is observed (sliced) according to some property or rule. The main purposes of slicing are to understand code dependencies, support debugging, and to manipulate code to introduce or eliminate some property. Slicing thus divides source code and then re-composes it in a different manner to add or remove some desired or undesired properties; slicing is also used for better understanding.

Our work also relates to the research on separation of concerns [13]. The aim of separation of concerns is to elicit modeling information or code that relates to individual concerns. For instance, a concern could be a non-functional requirement that has to be satisfied. By separating concerns, it becomes possible to manipulate them without affecting one another. Our approach is a natural complement to separation of concerns. We believe that it is possible to define scenarios based on concerns. By using our approach one can then find overlaps among those concerns.

The xlinkit approach [12] is based on XML technologies and provides a mechanism for generating consistency links between XML-based documents. New links (=traces) are generated for every model element satisfying some given consistency rules. However, consistency rules may be difficult to define and expensive to evaluate. Our approach does not require them.

8. CONCLUSION

This paper presented an approach to trace analysis. As a pre-requisite, it requires hypotheses on how model elements relate to source code. Test scenarios and an executable software system may be used to determine this automatically; however, developers may also define this input manually. Trace dependencies exist among model elements if their source code overlaps.

This paper is based on an existing technique that requires at most n input dependencies (model element to source code) to generate up to n^2 trace dependencies. The existing approach avoids the complex issues of syntactic and semantic differences among models. We demonstrated this on the state transition/class example where we never defined the semantics of the models.

The drawback of the previous approach was in assuming that it would be easy for developers to identify input dependencies and shared code. We eased this burden by allowing developers to capture input with uncertainties. Particularly, we allowed the grouping of model elements and the expression of input confidence (e.g., isAtLeast, isAtMost). Furthermore, we augmented trace analysis with the ability to identify shared code and inconsistencies. Our extended approach is more flexible and more precise at the same time. The only condition for using our approach is the availability of an executable and observable software system (or prototypes)

9. REFERENCES

[1] Abi-Antoun, M., Ho, J., and Kwan, J. Inter-Library Loan Management System: Revised Life-Cycle Architecture. Technical Report, University of Southern California, 1999.

[2] Boehm, B.W., Abts, C., Brown, A. W., et al: Software Cost Estimation with COCOMO II. New Jersey, Prentice Hall, 2000.

[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] Cox, L., Delugach, H. S., and Skipper, D.: "Dependency Analysis Using Conceptual Graphs," Proceedings of the 9th International Conference on Conceptual Structures, Palo Alto, CA, July 2001.

[5] Dohyung, K.: "Java MPEG Player," http://peace.snu.ac.kr/dhkim/java/MPEG/, 1999.

[6] Egyed A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering (TSE) 29(2), 2003, 116-132.

[7] Egyed, A. and Gruenbacher, P.: "Automating Requirements Traceability – Beyond the Record and Replay Paradigm," Proceedings of the 17th International Conference on Automated Software Engineering (ASE), Edinburgh, Scottland, UK, September 2002, pp.pp. 163-171.

[8] Gotel, O. C. Z. and Finkelstein, A. C. W.: "An Analysis of the Requirements Traceability Problem," Proceedings of the First International Conf. on Requirements Engineering, 1994, pp.94-101.

[9] 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

[10] Jackson, J.: "A Keyphrase Based Traceability Scheme," IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, 1991, pp.2-1-2/4.

[11] Kiczales, G. , Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J.: "Aspect-Oriented Programming," European Conference on Object-Oriented Programming (ECOOP), June 1997, pp.220-242.

[12] Nentwich C., Capra L., Emmerich W., and Finkelstein A.: xlinkit: a consistency checking and smart link generation service. ACM Transactions on Internet Technology (TOIT) 2(2), 2002, 151-185.

[13] Parnas D. L.: On the Criteria to be Used in Decomposing Systems into Modules. Comm. of the ACM 15(12), 1972, 1053-1058.

[14] Pinheiro F. A. C. and Goguen J. A.: An Object-Oriented Tool for Tracing Requirements. IEEE Software 13(2), 1996, 52-64.

[15] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[16] Snelting, G. and Tip, F.: "Reengineering Class Hierachies Using Concept Analysis," Proceedings of the ACM SIGSOFT Symp. on the Foundations of Software Engineering, November 1998, pp.99-110.