Exploiting Traceability Uncertainty between Architectural Models and Code

Achraf Ghabi
Johannes Kepler University (JKU)
Linz, Austria
achraf.ghabi@jku.at

Alexander Egyed
Johannes Kepler University (JKU)
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Documenting and maintaining the traceability between architectural models and code is one of the foremost challenges of model-based software development. Yet, traceability is rarely captured immediately while models and code co-evolve but usually recovered later. By then key people may have moved on or their recollection of facts may be blurred or inconsistent. In previous work, we proposed a language for capturing traceability that allows for uncertainty and incompleteness. This paper investigates this language on the unique properties that characterize model-to-code traceability. Our approach takes ambiguous, incomplete, and possibly incorrect assumptions about the traceability between model and code as input. It then validates the correctness of these assumptions and completes the input by inserting their logical consequences. This paper demonstrates the correctness and scalability of our approach which has been validated on several third-party software systems. Our approach is automated and fully tool supported.**

Keywords-traceability, model to code mapping, analysis

I. Introduction

Traceability is needed for both model and code understanding [21]; and it is vital for change impact analysis during maintenance [5]. Good tool support exists for recording traces, however, less so for understanding the relationship between architectural artifacts and code. Traceability is typically captured manually in form of trace matrices that cross-reference model elements and pieces of code (i.e., their classes or methods). The engineers' job is to fill in the fields of the matrix by deciding for each model element and piece of code separately whether or not the piece of code implements the model element ($n*m$ problem). Each decision is not trivial and there are many such decisions. Consider, for example, the ArgoUML system [36] (one of our study systems) with hundreds of model elements and tens of thousands of Java methods. A complete traceability matrix for the ArgoUML requires millions of decisions; one for every model element/Java method pair. The scalability implication is daunting [3]. And the traceability, once established, must be updated when the model and code change for it to remain consistent and useful [6]. Unfortunately, key personnel may have moved on or may not remember vital details. The engineers' knowledge on the model to code traces is likely to be outdated or incomplete and becomes worse over time [20].

This paper emphasizes on architectural model-to-code traceability for architecture elements which are realized in the code. *It is important to note that architectural models may address both the problem and solution domain. This work thus focuses only on those parts of the architectural model that are realized as code.* Here, architectural model-to-code traceability suffers from the general problem that traceability is rarely captured immediately while models and code co-evolve but usually recovered later. By then key people may have moved on or their recollection of facts may be blurred or inconsistent. This requires special language constructs to express uncertainty and incompleteness which current state-of-the-art in traceability does not do. Architectural model-to-code traceability, however, also suffers from a misconception, often implied in literature, where each piece of code (e.g., method or class) belongs uniquely to one architectural element only. The property of uniqueness is much more subtle. On one hand, there could be multiple architectural perspectives (i.e., models) for a software system (e.g., a structural one in form of a component diagram and a behavioral one in form of a statechart diagram). Here it is quite reasonable to assume that a given piece of code implements both the structure *and* the behavior – hence it implements multiple architectural elements. However, even in context of a single architectural perspective (e.g., a component diagram), uniqueness is a tricky property because on the code level it is quite reasonable to assume that code is either reused among components or directly referenced (e.g., the data models two components share if they are to communicate).

This paper adopts the traceability language for model-to-model traceability via code introduced in [16] and extends it for architectural models-to-code traceability considering some of the unique aspects of this domain and also providing a more effective data structure for capturing and maintaining traceability information. Our proposed approach allows traceability to be captured incomplete and it may contain typical uncertainties. An example of such a traceability uncertainty is that the engineer knows that some given piece of code may implement an architectural element; however, not whether this piece of code also implements other architectural elements; or whether other pieces of code also implement this architectural element. It would be wrong for a trace capture tool to force a precise input from the engineer in the face of such uncertainties.

The main benefit of our approach is that it allows the engineer to express traceability to the level of detail (completeness and/or certainty) he or she is comfortable with. This is contrary to existing techniques for trace capture, usually relying on trace matrices, where an engineer must express for each architectural model element and piece of code whether there is or there is no traceability. We believe

that our language is most useful for situations where multiple engineers collaborate such that each engineer provides individual input about traceability (incomplete and/or with uncertainties); yet, the combinations of this input allows for more precise and complete reasoning. Indeed, we will demonstrate that it is possible to automatically reduce, even resolve, the incompleteness and uncertainties by automatically inserting logical consequences of the engineers' input. And we will demonstrate that it is possible to automatically identify incorrectness where the input provides contradictory facts (it is often not trivial to recognize incorrectness). Finally, we will discuss correctness, scalability, and effectiveness on lessons we have learned from four case studies.

II. ILLUSTRATION

We use an illustration throughout this. While simple, this illustration allows us to address many of the uncertainty and incompleteness issues that characterize model-to-code traceability. The illustration in Figure 1 depicts a state transition diagram with four model elements {*select*, *play*, *playing*, *stop*} and five pieces of code – labeled by their short acronyms {A,B,C,D,E} (i.e., A could stand for a class or a method). This statechart diagram describes the behavior of a movie player [12]. The movie player supports *stop* and *play* transitions to/from the playing state. Selecting a new movie automatically starts the playing of the movie.

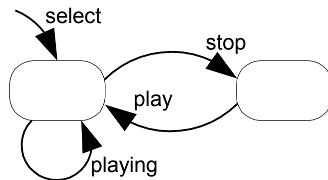

Figure 1. Illustration of State Transition Model

III. RELATIONSHIPS BETWEEN MODEL AND CODE

While it is becoming more common that developers create and use architectural descriptions, it is still not common to document where exactly a each architectural element is implemented in the source code. Knowing about traceability is important for code understanding and understanding the impact of a change (e.g., if a part of the architecture changes). The goal of this work is to help the engineer explore this relationship between model and code.

We refer to a piece of source code as a ***code element*** where the granularity of the code element is entirely user-definable. A code element could be a line of code, a method, a class, a package, or any other logical grouping (e.g., architectural component). We will discuss the implications of different granularity choices later. We presume that the code elements are disjoint in that every line of code must belong to at most one code element.

We refer to pieces of models (e.g., components, states, transitions) as *model elements*. is bidirectional. On the one side, we expect that a single model element is implemented in multiple code elements (one-to-many mapping) because model elements are typically higher-level descriptions of the implementation of the system and hence require arbitrary larger amounts of code to implement them. However, as was discussed in the introduction, this is not to say that each code element implements a single architectural model element only. It is possible that a given code element does implement a single model element only (uniqueness) but it is also possible that a given code element implements multiple model elements. Moreover, it is not always correct to assume that every code element must implement a model element. This assumption is true only if the model describes the entire software system. Models can be incomplete; either by choice or by omission and thus, a code element could also be not implementing any model elements.

Many kinds of architectural models exist. The above illustration is an example of a behavioral model but there are also structural models (e.g., component diagrams), usage scenarios and others. Thus models provide independent perspectives onto a software system – we speak of multiple ***perspectives*** or views [2, 20, 32]. Each perspective describes the software system from a different point of view. For example, the requirements perspective describes the software system independently from the component structure or the behavior (e.g., Figure 1). Perspectives may be at different levels of abstractions or variations at the same level of abstraction (i.e., separating the structure from the behavior). A code element may thus implement *multiple model elements of different perspectives*. Here the code element may be uniquely implementing a single model element within any given perspective.

While the model elements of different perspectives are independent, the model elements within a single perspective could be complementary. For example, in a component diagram, each component is expected to contribute a *unique capability* to the software system that is not being contributed by any other component of the *same* component diagram. The correctness of this is obvious through negation: if two distinct components of a diagram contribute the exact same then why are they different? In [16], we had assumed that every model element in a perspective must be implemented by some unique code. While this is true, it is at times hard (if not impossible) to separate the unique code pieces into separate code elements because of:

- interwoven features: different capabilities that are implemented in close proximity in the source code and thus always used together (e.g., a bank transaction feature interwoven with a logging feature)
- common functionality: a piece of code that provides application-specific services for different purposes (e.g., playing a movie which is invoked by both selecting a movie and un-pausing it).

IV. LANGUAGE FOR EXPRESSING TRACEABILITY

A. *Defining Precise Trace Information*

Existing state-of-the-art requires precise traceability information which is captured in form of a trace matrix. In our domain, such a trace matrix would identify the architectural model elements and code elements at a level of detail defined by the engineer. The traceability between a

model element *m* and a code element *c* would then be defined either as a *trace(m, c)* indicating that *c* is implementing *m*; or as a *no-trace(m, c)* indicating that *c* does not implement *m*. Establishing such traceability information requires a precise knowledge about each code element and model element individually. In addition to the high effort that this task demands, it is also potentially error-prone due to the large amount of precise knowledge needed (trace matrix has the size=#code element * #model elements). Individual engineers typically only have such precise knowledge (i.e. expertise) on parts of a system in which they have been personally involved with; or the parts of the architecture. Therefore their knowledge about the remaining system would be more imprecise and uncertain.

B. Expressing Uncertainty

In [16] we introduced a flexible language for defining hypotheses on how model elements are related to code elements and/or tests. This language recognizes that an engineer may understand some model-to-code relationships even though there may be uncertainties. ***The language allowed the engineer to express the certainties without having to make (wrong) assumptions about the uncertainties***. Two levels of uncertainties are supported by the language:

- Uncertainty through the Grouping of Elements: Engineers may know well the role of *groups* of model and/or code elements but they may not understand them individually. For example, one may know that the selection and subsequent playing {select, playing} of a movie is implemented in code elements {A, B, C}. Yet, one may not know which code elements belong to {select} or {playing} individually because they are implemented together.
- Uncertainty in the Exact Scope of a Trace: Engineers may be uncertain whether a given set of code elements implements a model element completely. Or they may be certain that the implementation is buried inside some code without knowing exactly where. The language allowed the engineer to qualify the model-to-code relationships through *implAtLeast*, *implAtMost*, and other constructs discussed later (see Section D).

C. Defining Basic Uncertainty

Each of the uncertainty constructs builds a relationship between a set of model elements and a set of code elements. Each of them should contain at least one element (i.e. relationships to/from empty set are not allowed). The construct is defined as *{m*} relationship {c*}* where *{m*}* is the set of model elements and *{c*}* is the set of code elements. The star symbol (*) in this notation expresses multiplicity in that m* stands for multiple model elements and c* for multiple code elements. The *relationship* term declares how the first set is related to the second one. We distinguish between four major relationships: *implAtLeast*, *implAtMost*, *implExactly*, and *implNot*. Such input expresses implementation of code but leaves a range of issues unspecified:

1) an individual code element in *{c*}* may or may not be implementing any model element in *m**.
2) the model elements in *{m*}* may be implemented by code other than *{c*}* (denoted as *C-{c*}* where *C* is the set of all code elements).
3) other model elements within the same modeling perspective (denoted as *P-{m*}* where *P* is the set of model elements in perspective) may be implemented by code in *{c*}*.

This basic language is sufficient to express arbitrary complex model-to-code relationships. Depending on the nature of the relationship, the proposed constructs hint a relationship between single model elements from *{m*}* and the given code elements in *{c*}* and/or between single code elements from *{c*}* and the given model elements in *{m*}*. Therefore, we declare logical units to express these relationships: on the one side, the *code element group* (CEG) is a group bundling one model element with a set of code elements, e.g., *ceg(play, {B,C})* expresses that the model element *play* is implemented by *B*, *C*, or both; on the other side, the *model element group* (MEG) is a group bundling one code element with a set of model elements, e.g., *meg(C, {select*, *playing})* expresses that the code element *C* is implementing *select*, *playing*, or both.

D. Defining Common Uncertainty Constructs

Human input on traceability is a mixture of certainties and uncertainties. It is straightforward to reason about the certainties. They are facts in a reasoning engine. It is more challenging to reason about uncertainties. Uncertainties provide a more flexible means for establishing input. However, uncertainties must be expressed as constraints on facts which require us to formalize these constraints and their logical consequences. This section discusses these logical consequences of uncertainties which are useful for assessing correctness and completeness of traceability and for better understanding trace granularity.

1) ImplAtLeast Input:

The input *{m*} implAtLeast {c*}* defines that the model elements in *{m*} are implemented by all of* the code elements in *{c*} and possibly more*. This input has a correctness constraint ensuring that every code element in *c** individually must be implementing a subset of *m**.

In the context of *implAtLeast* construct we derive a CEG for each of the model elements and a MEG for each of the code elements as follows:

```
forall m: implAtLeast.{m*}
     add ceg(m, implAtLeast.{c*})
forall c: implAtLeast.{c*}
     add meg(c, implAtLeast.{m*})
```

For example, let us consider the following input example:

```
Input 1: {select,playing} implAtLeast {A,C}
```

Each model element must be implemented by `A` and/or `C`. And each code element must be implementing `select` and/or `playing`. The corresponding MEGs and CEGs are:

- `meg (A, {select, playing}) and meg(C, {select, playing})`
- `ceg(select, {A, C}) and ceg(playing, {A, C})`

The MEGs describe a relationship between a single code element and multiple model elements. For example, `meg(A, {select, playing})` implies that code `A` must either implement the model elements `select` or `playing`. The "or" operator is a logical "or", implying that `A` may implement either `select` or `playing` or both `select` and `playing`. The CEGs describe a relationship between a single model element and multiple code elements. For example, `ceg(select, {A, C})` implies that the model element `select` must be implemented in either `A` or `C or A and C (logical "or" again)`. Note that this input expresses the certainty that each model element in *{c*}* must be implementing a subset of *{m*}*. But it also has uncertainties (2) and (3) above (e.g,. code "A" may implement any subset of model elements `{select, playing})`.

2) ImplAtMost Input:

The input *{m*} implAtMost {c*}* defines that the model elements in *{m*}* are implemented by *some of* the code elements in *{c*} but certainly not more*. This input has uncertainties (1) and (3) above. But it expresses the certainty that every other code element not in *{c*}* must not implement any model element in *{m*}*. It is important to understand what code elements are not implementing a model element because knowing that a code element is implementing a model element does not imply that it cannot be implementing another model element of the same perspective (shared code).

```
forall m:implAtMost.{m*} & c:C-implAtMost.{c*}
     add no-trace(m, c)
forall m: implAtMost.{m*}
     add ceg(m, implAtMost.{c*})
```

This input has a correctness constraint of the same nature discussed above and it has another correctness constraint in that there must not exist a code element that implements and does not implement the same model element.

For example, if `{stop} implAtMost {C,D}` then `stop` may not be implemented by code other than `C` or `D`:

- `ceg( stop, {C,D})`
- Certainties: `no-trace(stop, A); no-trace(stop, B); no-trace(stop, E)`

3) ImplNot Input:

The input *{m*} implNot {c*}* defines that the model elements in *{m*}* are not implemented by *any of* the code elements in *{c*}*. This input is a negation of the *implAtMost* input because *{m*}* is not implemented by *{c*}* implies that *{m*}* must *implAtMost C-{c*}* (the remaining code). But still, it is not legitimate to assume the implAtMost input as long as it has not been explicitly defined by the engineer. Furthermore, there is no need to derive MEG or CEG in the context of *implNot* construct. We could, however, generate precise traceability information indicating a no-trace between each model element in *{m*}* and each code element in *{c*}*.

```
forall m:implNot.{m*} & c:implNot.{c*}
          add no-trace(m, c)
```

4) ImplExactly Input

The input *{m*} implExactly {c*}* defines that every code element in *{c*}* implements one or more model elements in *{m*}* and that the model elements in *{m*}* are not implemented in any other code (*C-{c*}*), which allows us to define no-trace between each model element in *{m*}* and each code element in *C-{c*}*. We can also safely state that each code element in *{c*}* implements a subset of *{m*}*. But this does not mean that these code elements could not implement other model elements (*P-{m*}*) – uncertainty (3) above. This input has correctness constraints similar to the ones above and allows us to generate MEGs and CEG as previously discussed:

```
forall m:implExactly.{m*} & c:C-implExactly.{c*}
     add no-trace(m, c)
forall m:implExactly.{m*}
     add ceg(m, implExactly.{c*})
forall c:implExactly.{c*}
     add meg(c, implExactly.{m*})
```

For example, if `{play, playing} implExactly {B,C}` then we can generate two MEGs and two CEGs (e.g,. neither `play` nor `playing` may be implemented by code other than `B` or `C`). The implExactly input also implies a few certainties, such as no-trace(play, A) because if `play` must be implemented within `B` and `C`:

- `meg(B,{play,playing});meg(C,{play, playing})`
- `ceg(play, {B,C}); ceg(playing, {B,C})`
- Certainties: `no-trace(play, A); no-trace(playing, A); no-trace(play, D); no-trace(playing, D); no-trace(play, E); no-trace(playing, E)`

E. Footprint Graph

We capture both facts and constraints (certainties and uncertainties) in a graph structure, which we call the footprint graph. The graph contains a node for every code element (called CE nodes) and a node for each model element (called ME nodes). The connections between these nodes describe the certainties of the input (trace or no-trace) – and the certainties that are generated out of the logical consequences of the uncertainties. E.g., a *trace(m, c)* is depicted by a continues line between the ME node of m and the CE node of c. Analogically, *no-traces* are depicted by dashed lines. Furthermore, the graph contains nodes to capture model element groups (MEG nodes) and code element groups (CEG nodes). These two kinds of nodes describe the uncertainties of the input. The correctness constraints are inferred from these nodes. Note that the footprint graph in this paper is quite different from the same named graph in [16] which is due to the need to accommodate four different kinds of nodes compared to a single kind earlier.

```
Input 1: {select, playing} implAtLeast {A,C}
Input 2: {play, playing} implExactly {B,C}
Input 3: {stop} implAtMost {C,D}
```

For the simple illustration discussed in Section 2 and the three inputs discussed previously, Figure 2 depicts the nodes for all three inputs in a single graph structure – the footprint graph. The middle two columns depict the code elements (CE nodes) for A, B, C, D, and E; and the model elements (ME nodes) for select, playing, play, and stop. The left column depicts the model element groups (MEG) by connecting each set of model elements to the corresponding code element, and the right column depicts the code element

groups (CEG) by connecting each set of code elements to the corresponding model element. This graph structure depicts the certainties as connections between CE and ME nodes and uncertainties as connections between CE and MEG or ME and CEG. For scalability, the footprint graph structure grows linearly with the user input (#total nodes = #CE nodes + #ME nodes) .

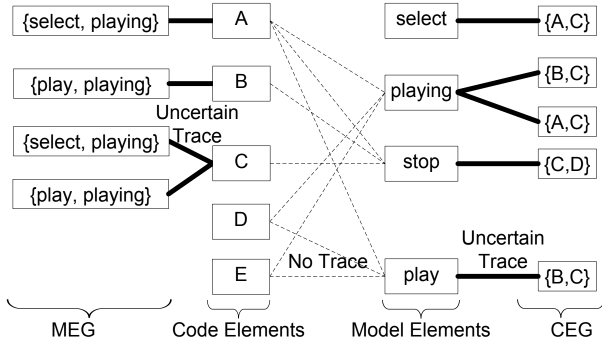

Figure 2. Footprint Graph from Input

The footprint graph is the foundation for automatic trace generation. During trace generation, the model elements in the graph are propagated from the CEG and MEG nodes (containing the uncertainties) to the CE and ME nodes (connected by the certainties). There are several such propagation rules discussed below ([16] supported only one of them).

F. Propagation Rules for Reducing Uncertainty

Consider the example in Figure 2 once again. The first input resulted in `meg(A, {select, playing})` implying that `A` must implement either `select` and/or `playing`. Then the third input resulted in `no-trace(play, A) and no-trace(playing, A)`. So if `A` is supposed to be implementing `{select, playing}` but `A` is not supposed to implement `playing` then clearly `A` must be implementing `select` – the only remaining model element in the MEG. Recall that the MEG defines a constraint over multiple model elements where at least one of these model elements has to be implemented by the code element. Uncertainties in a MEG can thus be resolved (or reduced) by eliminating those model elements that are implemented by the code:

```
if no-trace(m, c)
     forall ceg:CEG where c in ceg.{c*}
             ceg.{c*} := ceg.{c*}-c
     forall meg:MEG where m in meg.{m*}
             meg.{m*} := meg.{m*}-m
```

G. Propagation Rules for Suggesting Trace

Uncertainties in a CEG are resolved similarly. For example, the first input also resulted in `ceg(playing, {A,C})` implying that `playing` was supposed to be implemented in either `A` and/or `C`. Since `playing` was excluded from code element `A`, the CEG is left with only one code element, namely `C`. This remaining code element *must be implementing* `playing` for CEG to be satisfied.

```
if ceg.{c*}.size=1 then
     trace(ceg.m, ceg.{c*}.first)
     remove ceg
if (meg.{m*}.size=1) then
     trace(meg.{m}.first, meg.c)
     remove meg
```

Figure 3 depicts the footprint graph after the application of the propagation rules discussed above. Note that the certainty increased as the links between MEs and CEs increased while uncertainty decreased (fewer CEG and MEG nodes). The propagation rules are applied for as long as possible. The order in which the rules are applied is irrelevant.

H. Uniqueness

We previously discussed that the components in diagrams typically form perspectives. For example, the components in a component diagram form a perspective because each component is expected to contribute a unique capability to the system not contributed by another component *of the same perspective*. Or each transition in Figure 1 contributes a unique behavior to the system not contributed by another transition of the same figure. Knowledge on perspectives is very useful for identifying additional propagation rules [16]. Yet, we previously made the trivializing assumption that all model elements in every perspective must contribute something unique. We discussed earlier that this assumption is not always true due to the level of granularity in the source code. *The uniqueness property is thus an optional input that would help improve the reasoning about uncertainties if provided.*

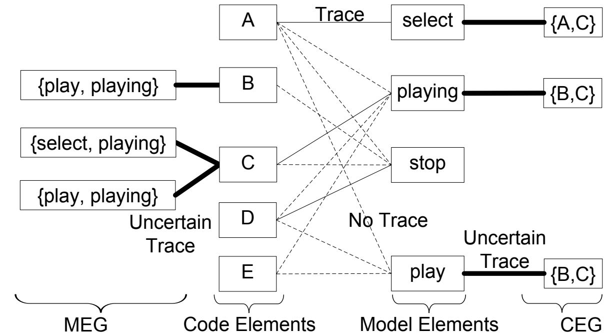

Figure 3. Footprint Graph after Propagation Rules

We thus added the ability to define uniqueness for CEG and MEG. For example, the third input `{stop} implAtMost {C,D}` could be annotated with the "unique" property. An engineer would do so if she is certain that `stop` is implemented in either `C` and/or `D`; and that some of this implementation is unique to `stop` (i.e., not shared with other model elements of the same perspective such as `play`, `playing`, or `select`).

If the `ceg(stop, {C,D})` has the uniqueness property then a code element can be removed in two ways: 1) by a no-trace between the mode element and the code element or 2) by the code element being shared among multiple model elements. Shared code is code that is implementing multiple model elements. Consequently, shared code is not uniquely implemented by a single model element. A model element is

unique to a code element *or* it has the potential to be unique if: (1) the code element implements at most the model element, and (2) the model element is contained in every MEG referenced by the code element.

To understand this, pay attention to the two MEG nodes in Figure 3. The `meg(C, {select, playing})` was added by input 1 and the `meg(C, {play, playing})` was added by input 2. It is incorrect to simply intersect the two sets to determine relationship. Thus while the two MEGs intersect in {playing}, this does not allow us to conclude that `C` must be implementing *playing* (although this is indeed correct in this example due to other input). However, *the intersection does allow us to reason about the unique relationship of a code element.*

If `C` were implemented by `playing` then both MEGs would be satisfied and `playing` would uniquely implement `C`. However, if `select` was implemented by `C` then only the first MEG would be satisfied. Thus, to satisfy both MEGs the code element would also have to be implementing a model element in the second MEG. In this case, the code element would be shared. So, *while we do not know who is implemented by* `C`*, we do know that only* `playing` *could be uniquely implemented by* `C`. Consequently, `C` cannot be used to satisfy the uniqueness property of the `ceg(stop, {C,D})` which requires `stop` to implement unique code in `C` and/or `D`. Therefore, `C` can be removed and the only remaining code element `D` must be the unique implementer of `stop` (for the CEG to be satisfied). We thus may include `stop` in `D` and we may safely exclude all other model elements of the same perspective from `D`. because any one of them implemented by `D` would violate the uniqueness property of the CEG. A simple property of uniqueness thus may have strong implications on traceability reasoning (reducing uncertainty). We omitted to algorithms due to brevity.

I. Correctness Constraints

Input given by the engineer may be partially/fully generated by hand and may be based on potentially outdated documentation or second-hand information (i.e., from a previous project members). It is important to provide correctness checks based on the consistency of the input. Fortunately, not every input combination is valid and our approach identifies four forms of input inconsistencies that indicate incorrect input. Do note that consistency does not imply correctness; however, with increasing quantity of input it becomes increasingly unlikely that the input remains consistent, especially if the input is provided by different engineers. The following demonstrates how our graph structure supports correctness checking.

(1) Every MEG must have at least one model element:

$$\forall_{meg \in MEG} meg.size > 0$$

A MEG is created if a code element is known to include one or more model elements (e.g., recall *implAtLeast*). Thus, it is invalid to have all model elements removed from a MEG. For example, such a violation occurs with the following input:

```
Input 4: {select} implNot {A}
```

Recall from Figure 3 that the `meg(A, {select, playing})` from input 1 was previously reduced to `trace(select, A)` because `playing` is not implemented by `A`. If now `select` is also not implemented by `A` then the MEG is left without a model element. In this case, input 1 could no longer be satisfied. Note that it is typically easy to see when two inputs conflict but it is hard to see conflicts among three or more inputs. *The example above is a conflict among inputs 1, 2, and 4 and not obvious to identify despite the small size of the illustration*.

(2) Every CEG must have at least one code element:

$$\forall_{ceg \in CEG} ceg.size > 0$$

A CEG is created if a model element is known to be implemented by one or more code elements. It is invalid to have all model elements removed from a CEG. Such a violation occurs with the input:

```
Input 5: {playing} implNot {C}
```

Recall from Figure 3 that the `ceg(playing, {A,C})` from input 1 was previously reduced to `trace(playing,C)` because playing was not implemented by `A`. If now playing is also not implemented by `C` then the CEG is left without a code element.

(3) Every model element must be imp. by some code:

Even if no CEG or MEG is violated, we must still make sure that every model element is implemented by some code (recall that our approach is applicable only for solution elements which are implemented in the code). This check is particularly useful for those model elements in perspectives for which no input was defined.

(4) A code element cannot be implementing and not implementing a model element at the same time:

A CE node contains the certainties of the input and the resolved uncertainties of the CEG and MEG nodes. These certainties should not conflict such that a code element be implementing and not implementing the same model element. Obviously, saying `{play} impl {A}` and `{play} implNot {A}` produces this kind of error. Note that our work in [16] supported this last correct constraint; however, none of the others.

J. Granularity Constraints

While software development standards mandate the establishment of traces between model and code, they do not define at what level of granularity (detail) these traces should be generated. For example, if the code is implemented in Java then the engineer has the choice to establish traceability between the model elements and the Java classes or the model elements and Java methods. It is also possible to establish the traceability to Java packages or its individual lines of code.

Obviously, the level of granularity vastly affects the cost of trace generation. In [17], we determined on three case studies (ArgoUML [36], Siemens Route Planning [22], Video on demand client [12]) that the input quantity of the model-to-class mappings was almost 10 times less than the input quantity of the model-to-method mapping; but 10

times more than the model-to-package mapping. This represents a significant cost factor since this ratio is roughly equivalent to the effort.

However, in [17], was discussed that a coarser granularity resulted in quality loss because functionality was grouped together that was separated on a finer granularity (i.e., we found a 16% increase in the false positives rate of traces based on their overlap on Java methods versus Java classes). Obviously, what granularity rate to choose depends on the needs of the traces and the effort one is willing to spend. But in the three case studies we evaluated, we found that the return on investment flattens out significantly when the granularity was finer than implementation classes (i.e., traces between model and methods/lines of code cost much more than was gained in quality).

Previously, we argued that the granularity should be staged depending on the importance of the model element. One may start off by defining the granularity on a coarser level (e.g., model to Java classes) and then refine key areas to a finer level of granularity (e.g., model to Java methods). Here we propose an additional avenue because we found that it is possible to define granularity constraints that tell (in some cases) and suggest (in other cases) which code elements to refine.

(1) Every correctness constraint a granularity constraint:

It must be mentioned that any of the four correctness constraints discussed above could be caused by coarse granularity. Recall that input 4 `{select} implNot {A}` caused a correctness violation because the code element excluded both model elements `select` and `playing` of the MEG. But what if code element `A` was too coarse grained and should have been broken down into methods, say `A1` and `A2`. The following input, on a finer level of granularity, resolves the conflict:

```
Input 1: {select, playing} implAtLeast {A1, C}
Input 4: {select} implNot {A2}
```

Correctness violations indicate problems where the input cannot be reconciled. Granularity issues thus may cause correctness violations because they might group code elements that should not belong together. Note that it is not necessary to refine the granularity level of all code elements. The correctness constraint identified the code element `A` as the offending place. A selected refinement of just `A` is thus sufficient to resolve the problem if it is the result of a granularity problem. Of course, some input may be incorrect irrespective of the granularity. Changing the granularity there would not resolve the problem.

(2) Every model element should have *unique* code:

Ideally every model element of a perspective should have unique code not shared with any other model element of the same perspective – if the level of granularity is fine enough. For example, the VOD system violated this constraint because the `select` model element was in fact invoking the `play` model element (i.e., select automatically started the movie if successful). The input below does not have a correctness violation but it does have a granularity warning:

```
Input6: {select,playing}implExactly{A,B,C}(unique)
Input7: {play,playing} implExactly {B,C}(unique)
```

This input expects unique code elements for `select`, `play`, and `playing`. However, `play` won't find unique code in either `B` or `C`. This issue is not a correctness error but an indication that the granularity of either {B} or {C} is wrong and may need to be refined. Not all granularity warnings can be resolved. We found situations where features in source code were interwoven to such a point where it was impossible to separate them.

(3) Every code element group with the uniqueness property should have unique code:

This applies to those CEG that have the uniqueness property. The uniqueness property implies that some but not necessarily all of the code elements must implement the model element uniquely.

(4) Every model element group with the uniqueness property should have unique code:

Same as granularity constraint (3) but for MEGs.

K. *Completeness Constraints*

Input that is correct is not necessarily complete. Recall that our input language allows for two degrees of uncertainties – partiality and cluster uncertainties. The propagation rules discussed above demonstrated how some uncertainties can be resolved. Yet, it must be stressed that the propagation rules must adhere to the logical consequences of the input. Likely not all input uncertainty can be resolved and it is useful to quickly identify those model elements and/or code elements that are still incomplete. For a model element to be complete, it must have traces and no-traces to all code elements:

$$complete(m) \Rightarrow \#trace(m) + \#no-trace(m) = \#C$$

The completeness of a model element can be determined for every model element separately. However, to determine the unique versus shared property of code, all code elements implementing a model element must be complete also. A code element implementing a model element *m* is complete if all other model elements of the same perspective *P-m* are either defined as trace or no-trace. Of course, if other model elements are tracing then the code element is shared; otherwise it is unique to the model element *m*.

V. VALIDATION

Our approach was evaluated in terms of its correctness, scalability, and effectiveness. The following presents results on four case studies (ArgoUML [36], Siemens Route Planning [17], Video on demand client [12], and USC Inter-Library Loan), the largest of which was the ArgoUML with over 28,000 methods in well above 1000 Java classes. The case studies involved a range of modeling perspectives (requirements, class diagrams, statechart diagrams, data flow diagrams) and two programming languages (Java, C++).

A. *Correctness*

The approach's correctness was evaluated informally by engineers (Siemens) and through extensive manual testing. In addition, we pair wise evaluated all combinations of the

four types of input (*implAtLeast*, etc…). Every input describes a relationship between a set of model elements and a set of code elements. Between any two inputs, the model elements of the one input may be a subset of the model elements of the other input – or it may be a superset, intersection, or not overlapping at all. Similarly, there are 4 scenarios on how the code elements of two inputs may overlap. Consequently, there are 4 * 4 * 5 * 5 = 400 possible input scenarios for two inputs. This evaluation confirmed that our approach produces correct results. However, it also allowed us to understand some of its limitations – particularly during constraint checking while deciding what is incorrectness and what is granularity; and during the resolution of constraint violations.

Due to the combinatorial explosion of multiple inputs (n inputs overlap in n^2 ways) it was impractical to provide "potential" feedback on incorrectness. An earlier version generated hundreds of potential errors about the ArgoUML system. In this paper, we thus defined correctness constraints that are guaranteed to be correct and we measured the success rate based on the 400 possible input scenarios. There, we found that 29% of all scenarios resulted in conflicts. Thus, given any two *incorrect* input rules, there is a 29% likelihood of us catching it.

Fortunately, here the combinatorial explosion is to our advantage. There are 100 pair wise relationships among 10 inputs and each pair wise relationship has a 29% of catching incorrectness. This observation of course assumes that all input scenarios are equally likely and they all contribute new facts – neither of which is true. However, this observation provides us with the confidence that incorrectness is more likely to be detected the more input is provided. On the four case studies, we found it virtually impossible to come up with a complete input that is internally inconsistent. The only exception was *incorrect but consistent* input. If an engineer has an incorrect but consistent understanding of the model-to-code mapping then no incorrectness may ever be detected. This case is rather unlikely if multiple engineers are involved in the creation of the input and/or if legacy input is being reused. Also, generated traces are potentially incorrect. As such there could be missing traces (false negatives) and existing but wrong traces (false positives) – depending on the correctness of the input. *If the input is guaranteed to be correct then our approach (1) will refine traces correctly and (2) will not generate incorrectness errors*. However, depending on the level granularity, our approach may not correctly identify which code is uniquely implementing a model element.

B. Scalability

The growth of the footprint graph is polynomial with the size of the model and code. A graph contains one node per code element and as many CEG and MEG as there are model elements and code elements per input. In the most extreme case, every input is about every model element and every code element. In this extreme case, the size of the graph will grow to:

$$size\ of\ graph = \#C + \#M + \#input * (\#C + \#M)$$

Which is the complexity O (#input * (#M + #C)) where #input represents the quantity of input, #M represents the number of model elements, and #C represents the number of code elements. In theory each input could be about all code elements and model elements but in practical cases we found that they are only about a small subsets of them. The largest systems we analyzed, the ArgoUML system, had well above 30,000 nodes (most of them being code elements at the granularity of methods) but our tool required less than a minute to convert the input into the footprint graph and propagate the rules for 38 requirements.

C. Effectiveness

Obviously, no single input resolves uncertainty. It is the combination of multiple inputs that does. We thus studied the effectiveness of any combination of 2 rules (out of the 400 possible input scenarios) in resolving uncertainties. Figure 4 demonstrates, for example, that two *implExactly* inputs (left) are twice as effective as two *implAtLeast* inputs in reducing/resolving uncertainties. On the other side, through the case studies we observed that *implAtLeast* rules are much easier to generate by the engineer than *implExactly* rules. To date, we have not been able to measure this cost/benefit trade-off. It is future work to find out whether, say, two inputs with uncertainties is more likely to be correct than a single input without uncertainties. In the same context, it would be interesting to find out whether input with uncertainties is proportionally cheaper to generate than input without uncertainties (i.e., proportionally to the completeness).

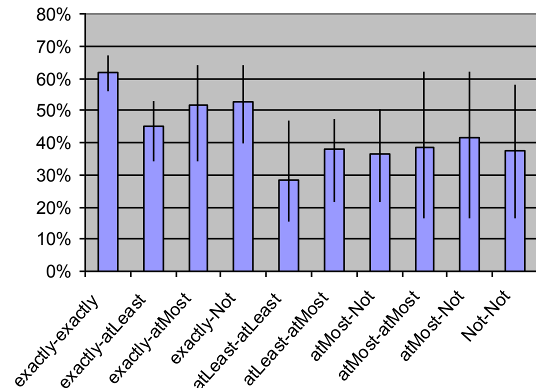

Figure 4. Effectiveness of Input in Reducing Uncertainties

Figure 4 also implies that the careful selection of input rules is as important as deciding on input types. For example, the maximum effectiveness of the *implAtLeast* inputs is hardly less than the minimum effectiveness of the *implExactly* input. Note that an input is less effective if it repeats some known facts. For example, if two inputs produce 4 CEG but two of these CEGs overlap then the input is repeating facts. Consequently, more input does not necessarily translate into more completeness. To this end, our approach reveals where more input is needed by investigating the MEG and CEG nodes (these nodes contain the uncertainties).

We also measured the relative importance of refinement (propagation rules) to reduce uncertainty and identify incorrectness. We found that refinement helped detecting 14% more incorrectness during the pair wise comparison.

D. *Resolving Incorrectness and Granularity*

A largely unsolved problem is that of resolving incorrectness and granularity problems. We can identify the input that is responsible for incorrectness and granularity problems. But this alone is not sufficient for the engineer to understand why there is a problem. It remains future work to provide support for resolving incorrectness and granularity problems.

VI. Open Issues

This work is limited to architectural model elements that are implemented in code. However, there are many other kinds of traceability that involve architecture models. Architects may need traceability between views; traceability to requirements; traceability to a design. There are subtleties in the kinds of traceability: an element may depend upon another; an element may implement another; an element may refine another. These kinds of traceability concerns are out of the scope of this work but are the focus of our future work. Also future work is an evaluation of our approach for different levels of input correctness and completeness; as well as scalability with increasing input sizes.

VII. Related Work

Research on traceability has progressed significantly and researchers have been developing automated approaches that go far beyond simple "recording and replaying" of trace links (which is still the level of support in many commercial tools). One of the earliest technologies for recovering requirements to code traces is Information Retrieval (IR) [9, 13] which identifies trace links based on naming similarities. Today, however, the traceability research goes beyond requirements-to-code traceability. There are many other kinds of approaches for the recovery of different types of trace links such as code and models [2, 18, 29], code and documentation [27], architecture and test cases [28], architecture and code [30], or features and code [11]. Researchers have proposed various techniques and heuristics to support the automation of trace recovery. Examples include event-based approaches [7], information retrieval [9, 13], feature location techniques [24], process-oriented approaches [34] scenario-based techniques [15], or rule-based methods [37]. This list of technologies recovers certain types of traces, for certain types of artifacts, at certain times. Although advances have been made to automatically recover links, trace capture remains a human-intensive activity [20, 26, 31]

The approaches of Haumer et al. [21], Jackson [23], and Cox-Delugach [10] 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. Concept analysis has been used in concert with manual input to provide a structured way of grouping traces. These groupings can then be formed into a concept lattice that is similar in nature to our footprint graph – but not as scalable [24]. Pinheiro and Goguen [33] approached traceability by devising 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 emphasizes on requirements. Antoniol et al. discuss a technique for automatically recovering traceability links between object-oriented design models and code based on determining the similarity of paired elements from design and code [2]. Spanoudakis et al. [37] have contributed a rule-based approach for automatically generating and maintaining traceability relations (between organizational models specified in i* and software systems models represented in UML). In the Goal-Centric Traceability (GCT) approach, Cleland-Huang et al. model non-functional requirements and their interdependencies as soft-goals in an Interdependency Graph. In their approach a probabilistic network model is used to retrieve links between classes affected by a functional change and elements within the graph [8]. A forward engineering approach is taken by Richardson and Green [35] in the area of program synthesis. Traceability relations are automatically derived between parts of a formal specification and parts of the synthesized program.

This proposed work is not the first work that recognizes the value in combining model dependencies (some limited types thereof) [8, 14]. However, to the best of our knowledge thus far nobody has tried to integrate and reason about many dimensions of model dependencies in such a rigorous, formal, and precise manner as we are proposing here. Also, the issues of uncertainties discussed in this work have not been explored in related work to the best of our knowledge. It is also important to note that traceability approaches typically do not provide explicit support for trace utilizations such as impact or coverage analysis. They rather provide general purpose features to create reports or query traceability information. Researchers have been proposing techniques to improve support for important tasks such as analyzing change impacts [1, 4, 25, 38] or understanding the conflict and cooperation among requirements [19]. There is however very little literature on the quality implications of trace links during such utilizations. As elsewhere, the utility of trace links decreases when the trace quality decreases. However, today, we have no understanding on how strong this effect is.

VIII. Conclusion

This paper presented an extension to our approach to trace discovery and validation. Our approach expects the engineer to define assumptions on architectural model-to-code traces (with incompleteness and uncertainties) and it then analyzes the correctness of these assumptions and is capable of resolving uncertainties. It must be noted that our approach does not "invent" traces. It discovers them based on the logical consequences of the assumptions provided. The ability to detect incorrectness protects the engineer from making errors. This is particularly important if the input was generated "after the fact" (after key people have moved on or

may have forgotten vital details), if the input was generated by different people (with inconsistent interpretations of traces), or if legacy traceability was reused (previously generated but no longer up-to-date) – as is typical during software maintenance.

IX. ACKNOWLEDGEMENT

This work was funded by the Austrian Science Fund (FWF) under agreement P23115-N23.

REFERENCES

[1] F. Abbattista, F. Lanubile, G. Mastelloni, and G. Visaggio. An experiment on the effect of design recording on impact analysis. In *Int. Conf. on Software Maintenance*, p. 253-259, September 1994.

[2] G Antoniol. Design-code traceability recovery: selecting the basic linkage properties. *Science of Computer Programming*, 40(2-3):213–234, July 2001.

[3] A. Bianchi, A.R. Fasolino, and G. Visaggio. An exploratory case study of the maintenance effectiveness of traceability models. In *Proc.8th Int. Workshop on Program Comprehension*, p. 149-158, Ireland, 2000.

[4] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *Proceedings of the Int. Conf. on Software Maintenance*, p. 256, USA, 2003.

[5] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. SÃ³wka. Automated impact analysis of UML models. *J. Syst. Softw.*, 79(3):339–352, 2006.

[6] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code, *SIGPLAN Notes*, 34(10):325–339, 1999.

[7] J. Cleland-Huang, C. K. Chang, and M. Christensen. Event-Based traceability for managing evolutionary change. *IEEE Trans. Softw. Eng.*, 29(9):796–810, September 2003.

[8] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. . *27th Int. Conf. on Software Engineering, ICSE*, p. 362–371, May 2005.

[9] J. Cleland-Huang, R. Settimi, E. Romanova, B. Berenbach, and S. Clark. Best practices for automated traceability. *Computer*, 40(6):27–35, June 2007.

[10] L. Cox, D. Skipper, and Harry S. Delugach. Dependency analysis using conceptual graphs. *In 9th Int. Conf on Conceptual Structures, 2001*.

[11] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Proceedings of the 22nd Int. Conf. on Automated software engineering*, ASE '07, p. 254–263, New York, NY, USA, 2007. ACM.

[12] K. Dohyung. MPEG player. http://peace.snu.ac.kr/dhkim/java/MPEG/.

[13] C. Duan and J. Cleland-Huang. Clustering support for automated tracing. In *Proceedings of the 22nd Int. Conf. on Automated software engineering*, ASE '07, p. 244–253, New York, NY, USA, 2007. ACM.

[14] M. Eaddy, A. V. Aho, G. Antoniol, and Y-G. Gueheneuc. CERBERUS: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *The 16th Int. Conf. on Program Comprehension*, p. 53–62, Amsterdam, The Netherlands, June 2008.

[15] A. Egyed. A Scenario-Driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng.*, 29(2):116-132, 2003.

[16] A. Egyed. Resolving uncertainties during trace analysis. In *12th Int. Symposium on Foundations of Software Engineering*, (SIGSOFT/FSE), p. 3–12, New York, USA, 2004. .

[17] A. Egyed, S. Biffl, M. Heindl, and P. Grünbacher. Determining the cost-quality trade-off for automated software traceability. In *20th Int. Conf. on Autom. Softw. Eng.*, p. 360–363, USA,2005. .

[18] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. *17th Int. Conf. on Automated Software Engineering*, p. 163–171. 2002.

[19] A. Egyed and P. Grünbacher. Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. IEEE *Software*,21(6):50–58, 2004.

[20] O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE Int. Conf. on Requirements Engineering*, p. 94-101, USA, 1994.

[21] P. Haumer, K. Pohl, K. Weidenhaupt, and M. Jarke. Improving reviews by extending traceability. In *Proceedings of the 32nd Annual Hawaii Int. Conf. on System Sciences*, 1999.

[22] M. Heindl and S. Biffl. A case study on value-based requirements tracing. In *Proceedings of the 10th European software engineering Conf.*, ESEC/FSE-13, p. 60-69, New York, NY, USA, 2005. ACM.

[23] J. Jackson. A keyphrase based traceability scheme. In *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, p. 2/1-2/4, December 1991.

[24] R. Koschke and J. Quante. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM Int. Conf. on Automated software engineering*, page 86, Long Beach, CA, USA, 2005.

[25] M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic analysis of the impacts of changes to Object-Oriented software. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, TOOLS '00, page 61, Washington, DC, USA, 2000. IEEE Computer Society.

[26] M. Lindvall and K. Sandahl. *Practical Implications of Traceability*. 1996.

[27] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th Int. Conf. on Software Engineering*, ICSE '03, p. 125–135, Washington, DC, USA, 2003.

[28] H. Muccini, P. Inverardi, and A. Bertolino. Using software architecture for code testing. *IEEE Transactions on Software Engineering*, 30(3):160–171, March 2004.

[29] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd Symposium on Foundations of software engineering*, p. 18–28, New York, NY, USA, 1995.

[30] L. G. Murta, A. Hoek, and C. M. Werner. Continuous and automated evolution of architecture-to-implementation traceability links. *Autom. Software Eng.*, 15(1):75–107, 2008.

[31] C. Neumuller and P. Grunbacher. Automating software traceability in very small companies: A case study and lessons learned. In *21st Int. Conf. on Automated Software Engineering*, p. 145–156, Sep. 2006.

[32] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[33] F. A. C. Pinheiro and J. A. Goguen. An Object-Oriented tool for tracing requirements. *IEEE Softw.*, 13(2):52–64, 1996.

[34] K. Pohl. PRO-ART: enabling requirements pre-traceability. In *Proceedings of the Second Int. Conf. on Requirements Engineering*, p. 76–84, April 1996.

[35] J. Richardson and J. Green. Automating traceability for generated software artifacts. In *Proceedings of the 19th IEEE Int. Conf. on Automated software engineering*, ASE '04, p. 24–33, Washington, DC, USA, 2004. IEEE Computer Society.

[36] J. Robins. ArgoUML. http://argouml.tigris.org/.

[37] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[38] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering* 29(6):49