The Journal of Systems and Software 108 (2015) 178–192

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Exploiting traceability uncertainty among artifacts and code

Achraf Ghabi*, Alexander Egyed

Johannes Kepler University, Altenbergerstraße 69, 4040 Linz, Austria

ARTICLE INFO

Article history:
Received 14 April 2014
Revised 29 May 2015
Accepted 16 June 2015
Available online 26 June 2015

Keywords:
Traceability
Artifacts to code mapping
Analysis

ABSTRACT

Traceability between software development artifacts and code has proven to save effort and improve quality. However, documenting and maintaining such traces remains highly unreliable. Traceability is rarely captured immediately while artifacts and code co-evolve. Instead they are recovered later. By then key people may have moved on or their recollection of facts may be incomplete and inconsistent. This paper proposes a language for capturing traceability that allows software engineers to express arbitrary assumption about the traceability between artifacts and code – even assumptions that may be inconsistent or incomplete. Our approach takes these assumptions to reasons about their logical consequences (hence increasing completeness) and to reveal inconsistencies (hence increasing correctness). In doing so, our approach's reasoning is correct even in the presence of known inconsistencies. This paper demonstrates the correctness and scalability of our approach on several, large-scale third-party software systems. Our approach is automated and tool supported.

1. Introduction

Traceability is very important during software development, especially for change impact analysis (Briand et al., 2006; Mäder and Egyed, 2011) during the maintenance stage (Haumer et al., 1999). Empirical evidence suggests that requirements to code traces can make bug fixes and features extensions 20–30% faster and over 50% more correct (Briand et al., in press; Mäder and Egyed, 2011). These benefits are substantial and accentuate that traceability should play a major role in the software engineering life cycle. Existing commercial tools typically support the recording of traces but not necessarily their creation or maintenance.

It is presumed that software engineers 'know' the traces between software artifacts (e.g., requirements or model elements) and code. Existing tools merely record them (Egyed and Grunbacher, 2002) – typically using a trace matrix (TM) that cross-reference artifacts at the level of granularity the engineers chose (e.g., requirements to classes vsrequirements to methods traces). The engineers' job is to manually fill in the fields of the matrix by deciding for each cross-reference whether or not the element on the one side, say a requirement, is implemented by the element on the other side, say a method. A trace matrix thus reveals that traceability is of quadratic complexity: $a*c$ for a artifacts (e.g., requirements) and c code elements (e.g., methods). Each cell in a trace matrix requires a non-trivial, human decision. Consider, for example, the Gantt Project system (GAN, 2014) (one of our study systems) with hundreds of artifact elements and thousands of Java methods. A complete traceability matrix for the Gantt Project system requires tens of thousands of decisions; one for every model element/Java method pair. The scalability implication is daunting (Bianchi et al., 2000). Once established, the traceability must be kept up-to-date while the software artifacts and/or the code changes (Clarke et al., 1999) – to remain consistent and useful.

Yet, traceability cannot be captured or maintained by a single engineer because in any complex engineering effort engineers have partial knowledge only. Traceability is thus a collaborative process that involves many engineers. Moreover, traceability is a mostly manual process (automation are mostly limited to information retrieval discussed later). Given that traceability is also of non-linear complexity, it should not surprise that there is never a guarantee of correctness or completeness. Naturally this is a problem because the aforementioned studies on the benefits of traces (Briand et al., in press; Mäder and Egyed, 2011) presume correctness and completeness. Now consider that today most engineering projects do not even capture traceability (upfront). Rather, they capture it at later stages (after system completion) or never in which case this knowledge remains in the heads to the engineers who built the system. Unfortunately, during the development of a system and after its delivery to the client, key personnel may move on. Even if they stay, it is well documented that the engineers' recollection of artifacts and code fades over time – and with it the memory of traceability (Gotel and Finkelstein, 1994). However, it is exactly here that traceability is most needed.

* Corresponding author. Tel.: +4373224684388.
E-mail addresses: a@ghabi.net (A. Ghabi), alexander.egyed@jku.at (A. Egyed).

http://dx.doi.org/10.1016/j.jss.2015.06.037
0164-1212/

Explicit traceability capture is thus a pre-requisite to principled software engineering. This paper introduces a language and approach that allows engineers to express traceability at any level of detail, completeness, certainty, and correctness. An example of a traceability uncertainty is if the engineer remembers that a given requirement is implemented in some set of classes but not exactly which ones of them. It would be wrong for a trace capture tool to force a precise input from an engineer in the face of such uncertainty. Yet, if multiple engineers input partially uncertain traceability then it is possible to combine this knowledge for a more complete understanding. We will demonstrate that it is possible to automatically reduce, even resolve, some uncertainty by automatically inserting logical consequences of the input provided by the engineers. As this example shows, our approach is most useful for situations where multiple engineers provide input about traceability. Yet, traceability provided by different engineers may not be consistent. We will demonstrate that it is possible to automatically identify incorrectness where the input provided by engineers is contradictory. But most significantly, we will demonstrate that this automation is correct even in the presence of inconsistent input.

This paper combines our findings from three conference papers where we described the traceability language for model-to-code traceability (Egyed, 2004), an effective reasoning mechanism that is able to check correctness (Ghabi and Egyed, 2012), and managing inconsistencies in SAT problems with HUMUS (Nöhrer et al., 2012). The added value is in (a) providing a scalable, precise basis for reasoning based on SAT solvers; (b) more numerous and larger empirical evaluations; (c) a broadened scope that covers requirements, model elements and code; and (d) the integration of HUMUS and SAT for correct reasoning in context of potentially inconsistent traceability.

2. Illustration

We use the illustration of a video-on-demand system (VOD) (Dohyung) throughout this work to explain many of the uncertainty and incompleteness issues that characterize artifact-to-code traceability. In Fig. 1 we depict a state transition diagram on the left side and a table of requirements on the right side. The state transition diagram models the behavior of the VOD system. The table of requirements on the right side of Fig. 1 is an abbreviated documentation of the requirements implemented in VOD. Together, these two diagrams depict the many artifacts that engineers may want to trace to the code. For example, each requirement (i.e. row) in the table is an artifact that should be implemented somewhere in the code. The same is true to for the state transitions. For the sake of brevity we will be referring to the requirements and state transitions by their IDs: e.g., R1 or S4.

The VOD is a real albeit smaller system implemented in Java. For the sake of brevity we abstract the implementation into five pieces of code – labelled by their short acronyms {A, B, C, D, E}. Each of them stands for a set of Java classes.

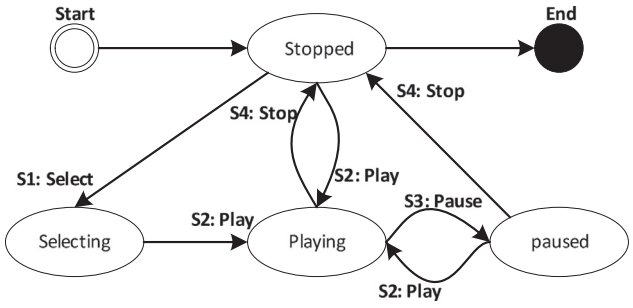
3. Artifacts and code relationships

While it is common that engineers create and use artifact descriptions, it is still not common to document where exactly each artifact is implemented in the source code or how it is related to other software development artifacts. Knowing about traceability is important for understanding complex systems and understanding the impact of a change (e.g., if a part of the requirements changes how would it impact the implementation?). The goal of this work is to help the engineer explore this kind of relationship between software development artifacts and the code. A software development artifact could be any common artifact used during the development and/or maintenance of a software project such as UML model, use cases, or requirements definition.

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 the same line of code may not belong to more than one code element.

We refer to individual requirements, states transitions, etc as artifact elements. Here also the granularity is arbitrary user definable. For example, we could trace the entire state transition diagram to code or we could trace its individual states and transitions. The relationship between artifact elements and code elements is bidirectional. We expect that a single artifact element is implemented in multiple code elements (one-to-many mapping) because artifact elements are typically higher-level descriptions of the implementation of the system. Hence they are expected to require larger amounts of code to implement them. However, a single code element may also implement multiple artifacts (particularly, if the granularity of code elements if coarse). Moreover, it is not correct to assume that every code element must implement an artifact element. This assumption is true only if the artifact elements describe the entire software system. Artifacts (e.g. models) can be incomplete either by choice or by omission. For example, the state transition diagram in Fig. 1 is by no means complete and hence not all code will trace to it.

Fig. 1 includes a state transition diagram (which is a behavioral model) and it includes also a list of requirements. Those artifacts provide independent perspectives onto a software system – we speak of multiple perspectives or views (Antoniol, 2001; Gotel and Finkelstein, 1994; Parnas, 1972). Each perspective describes the software system from a different point of view. For example, the state transition perspective describes the software system independently from the requirements but there are clearly overlaps. R4 about stopping the playback, for example, is also implemented in the state transition diagram through various transitions. Perspectives may be at different levels of abstraction (i.e., separating the structure from the behavior). A code element may thus implement artifact elements of different perspectives. For example, whatever code implements the stopping of a playback implements both the stop transitions and the stop requirement.



Req. ID	Name of Requirement
R1	Select a Movie
R2	Play a Selected Movie
R3	Pause Playing
R4	Stop Playing

Fig. 1. Illustration System: Video On Demand (VOD).

4. Language for expressing traceability

4.1. Precise trace information

Existing state-of-the-art requires precise traceability information which is often captured in form of a trace matrix (TM). Such a trace matrix would identify the artifact elements and code elements at a level of granularity defined by the engineer. Of course, precise trace information among some cells in a trace matrix may exist and engineers should still be able to capture them. In our language, the traceability between an artifact element *ae* and a code element *ce* would then be defined either as a *trace*(*ae, ce*) indicating that *ce* is implementing *ae*; or as a *no-trace(ae, ce)* indicating that *ce* does not implement *ae*. Establishing such traceability information requires a precise knowledge about each code element and artifact element individually. Typically, engineers have such precise knowledge (i.e. expertise) on the parts of a system which they have been personally involved with. However, they may also have knowledge about other parts of that system though perhaps less precise or certain.

4.2. Expressing uncertainty

In interviews and surveys with researchers and practitioners (Egyed et al., 2010; Mäder and Egyed, 2011; Mäder and Egyed, 2012) we identified a range of uncertainty scenarios. Subsequent study revealed that these uncertainties were nearly always the result of not knowing the role of individual artifacts or code in groups thereof. For example, one may know that the selection and subsequent playing (group denoted as {select, playing}) of a movie is implemented in code elements {A, B, C}. Yet, one may not know which part of code {A, B, C} belongs to select and which part belongs to playing individually. In Ghabi and Egyed (2012), we introduced the concept of groupings of artifact and code elements to support grouping uncertainties. The code element group (*CEG*) is a group bundling one artifact element to a set of code elements. For example, *ceg*(*R*2, {*B, C*}) expresses that the artifact element *R*2 is implemented by *B, C*, or both. Furthermore, the artifact element group (*AEG*) is a group bundling one code element with a set of artifact elements, e.g., *aeg*(*C*, {*R*1, *R*2}). It expresses that the code element *C* is implementing requirements *R*1, *R*2, or both. Indeed, *trace, no-trace, ceg*, and *aeg* are the four basic constructs of our approach through which all uncertainties are expressible. We do not argue that these are the most common constructs but rather they are the most simplistic constructs to express traceability certainties and uncertainties. However, more complex situations are awkward to express in form of *ceg* and *aeg*. Our approach thus also defines a higher-level language which describes situations we encountered in practice and can be broken down to the four basic constructs for reasoning. These are discussed next.

4.3. Language for expressing uncertainty

Engineers could provide input in form of the four constructs discussed above. In addition, our approach supports artifact to code relationships for more complex but common situations. These relationships are denoted as {*ae*∗}*relationship*{*ce*∗} where {*ae*∗} refers to the set of artifact elements, {*ce*∗} to the set of code elements, and relationship to the situation (e.g., implAtLeast, implAtMost, implExactly, or implNot). For example, one may know that {playing} is definitely implemented in code element {A} but it could be implemented in other code elements also: hence, {playing} implAtLeast {A}. The star symbol (∗) in this notation expresses multiplicity in that *ae*∗ may stand for multiple artifact elements or *ce*∗ for multiple code elements. We denote *CE* as the set of all code elements and AE^P as the set of all artifact elements within a given perspective *P* where the perspective is entirely user definable (e.g., requirements, architecture, or state chart). The statement $CE - \{ce*\}$ identifies all the code elements other than those identified in {*ce*∗} (i.e., a relationship usually affects a set of code elements {*ce*∗} but often also the complementary set of other code elements $CE - \{ce*\}$). Likewise, the statement $AE^P - \{ae*\}$ identifies all artifact elements in a given perspective other than those identified in {*ae*∗}. Note that this language was already introduced in Egyed (2004) in principle. In Ghabi and Egyed (2012), we then added a formal basis for this language based on the four basic constructs discussed above and guidance in form of consistency, completeness, and granularity constraints. This paper uses the formal basis provided there but provides a new SAT-based realization thereof. The key benefit of this realization is that it no longer fails in the presence of inconsistencies. Recall that inconsistencies are the result of contradictory input which is the norm if traceability originates from multiple engineers. Inconsistencies normally confuse a reasoning engine. However, this paper adds HUMUS as another technology to correctly isolate inconsistent input from reasoning to ensure that it no longer affects the correctness of the results. This paper also provides extensive empirical evaluation on the scalability and effectiveness of our approach (discussed later).

4.4. Defining common uncertainty constructs

We found that engineers provide a mixture of certainties and uncertainties as input to traceability. 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 flexible means for establishing input. Therefore, uncertainties must be expressed as constraints on facts which require us to formalize these constraints and their logical consequences. Based on our observation of engineers (Egyed et al., 2010; Mäder and Egyed, 2011; Mäder and Egyed, 2012) describing their certain and uncertain knowledge about traceability, we identified four common relationships covering the expression of the observed scenarios. Note that these relationships are not the only ones that exists but we found them to be applicable in many situations. Additional relationships may be defined as needed. This section discusses the different types of relationships and the logical consequences of uncertainties which are useful for assessing correctness and completeness of traceability and for better understanding trace granularity.

4.4.1. ImplAtLeast input

The input {ae∗} implAtLeast {ce∗} defines that the artifact elements in {*ae*∗} are implemented by all of the code elements in {*ce*∗} and possibly more. An engineer may want to use this relationship if s/he is certain that the given code elements *ce*∗ implement the given artifact elements *ae*∗ but s/he is not certain as to whether other code may also implement the given artifact elements (e.g., the engineer may have been in charge of some code that partially implemented a feature). This input not only provides some facts but it also implies a correctness constraint ensuring that every code element in *ce*∗ individually must be implementing a subset of *ae*∗. In the context of the implAtLeast construct, we derive a *CEG* for each of the artifact elements and an *AEG* for each of the code elements as follows:

```
forall ae : implAtLeast. {ae∗}
    add ceg (ae, implAtLeast. {ce∗})
forall ce : implAtLeast. {ce∗}
    add aeg (ce, implAtLeast. {ae∗})
```

For example, let us consider the following input example about requirements *R*1 and *R*3 in Fig. 1:

Input 1 : {R1, R3} implAtLeast {A, C}

Each requirement (i.e. artifact element) must be implemented by *A* and/or *C*. And each code element must be implementing *R*1 and/or *R*3. The corresponding *AEG*s and *CEG*s are:

- *aeg*(*A*, {*R*1, *R*3}) and *aeg* (*C*, {*R*1, *R*3})
- *ceg*(*R*1, {*A, C*}) and *ceg* (*R*3, {*A, C*})

As such, *aeg*(*A*, {*R*1, *R*2}) implies that code *A* must either implement the artifact elements *R*1 or *R*2. The "or" operator is a logical "or", implying that *A* may implement either *R*1 or *R*2 or both *R*1 and *R*2. The *CEG*s describe a relationship between a single artifact element and multiple code elements. For example, *ceg*(*R*1, {*A, C*}) implies that the artifact element *R*1 must be implemented in either *A* or *C* or *A* and *C* (logical "or" again). Note that this input expresses the certainty that each artifact element in {*ce*∗} must be implementing a subset of {*ae*∗}. But it also has uncertainties. The artifact elements in{*ae*} may be implemented by code other than {*ce*}(denoted as $CE - \{ce\}$ where CE is the set of all code elements) – in the following referred to as Uncertainty (1). Moreover, other artifact elements within the same artifact/perspective (denoted as $AE^P - \{ae\}$ where AE^P is the set of artifact elements in a perspective) may be implemented by code in ce – in the following referred to as Uncertainty (2). For example, code *A* may implement any subset of artifact elements {*R*1, *R*3}.

4.4.2. ImplAtMost input

The input {ae∗} implAtMost {ce∗} defines that the artifact elements in {*ae*∗} are implemented by some of the code elements in {*ce*∗} but certainly not more. This input has Uncertainty (2) above. Moreover, an individual code element in ce may or may not be implementing any artifact element in ae – in the following referred to as Uncertainty (3). But this input also expresses the certainty that every other code element not in {*ce*∗} must not implement any artifact element in {*ae*∗}. Note that it is also important to understand what code elements are not implementing an artifact element because knowing that a code element is implementing an artifact element does not imply that it cannot be implementing another artifact element of the same perspective. An engineer may use this relationship if it is known where roughly the given artifact elements are implemented but the details are unknown (e.g., the engineer may know that an artifact element is implement in class *A*, but not exactly which methods of that class).

```
forall ae : implAtMost. {ae∗}
    & ce : CE−implAtMost. {ce∗}
  add no−trace (ae, ce)
forall ae : implAtMost. {ae∗}
  add ceg (ae, implAtMost. {ce∗})
```

For example, if {R4} implAtMost {C, D} then *R*4 is either implemented in *C*, or *D*, or *C* and *D*; and *R*4 may not be implemented by code other than *C* or *D*:

- *ceg*(*R*4, {*C, D*})
- Certainties: *no-trace*(*R*4, *A*); *no-trace*(*R*4, *B*); *no-trace*(*R*4, *E*)

4.4.3. ImplNot input

The input {ae∗} implNot {ce∗} defines that the artifact elements in {*ae*∗} are not implemented by any of the code elements in {*ce*∗}. This input is a negation of the implAtMost input because {*ae*∗} is not implemented by {*ce*∗} implies {ae∗} implAtMostCE−{ce∗} (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 *AEG* or *CEG* in the context of the implNot construct. We could, however, generate precise traceability information indicating a *no-trace* between each artifact element in {*ae*∗} and each code element in {*ce*∗}.

4.4.4. ImplExactly input

The input {ae∗} implExactly {ce∗} defines that every code element in {*ce*∗} implements one or more artifact elements in {*ae*∗} and that the artifact elements in {*ae*∗} are not implemented in any other code ($CE - \{ce*\}$), which allows us to define no-trace between each artifact element in {*ae*∗} and each code element in $CE - \{ce*\}$. We can also safely state that each code element in {*ce*∗} implements a subset of {*ae*∗}. But this does not mean that these code elements could not implement other artifact elements ($AE^P - \{ae*\}$) – Uncertainty (2) above. This input has correctness constraints similar to the ones above and allows us to generate *AEG*s and *CEG*s as previously discussed:

```
forall ae : implExactly. {ae∗}
    & ce : CE−implExactly. {ce∗}
  add no−trace (ae, ce)
forall ae : implExactly. {ae∗}
  add ceg (ae, implExactly. {ce∗})
forall ce : implExactly.{ce∗}
  add aeg (ce, implExactly. {ae∗})
```

For example, if {R2, R3} implExactly {B,C} then we can generate two *AEG*s and two *CEG*s (e.g,. neither *R*2 nor *R*3 may be implemented by code other than *B* or *C*). The implExactly input also implies a few certainties, such as *no-trace*(*R*2, *A*) because if *R*2 must be implemented within *B* and *C*:

- *aeg* (*B*, {*R*2, *R*3}); *aeg*(*C*, {*R*2, *R*3})
- *ceg* (*R*2, {*B, C*}); *ceg*(*R*3, {*B, C*})
- Certainties: *no-trace (R2, A); no-trace(R3, A); no-trace(R2, D); no-trace(R3, D); no-trace(R2, E); no-trace(R3, E)*

4.4.5. 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 artifact element (called AE-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*(*ae, ce*) is depicted by a continues line between the AE-node of '*ae*' and the CE-node of '*ce*'. Analogically, *no-traces* are depicted by dashed lines. Furthermore, the graph contains nodes to capture artifact element groups (AEG-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 visual footprint graph in this paper is quite similar to the graph introduced in Ghabi and Egyed (2012). However, the implementation does not use this graph (it is for illustration purposes) but instead is based on SAT expressions.

```
Input 1 : {R1, R3} implAtLeast {A, C}
Input 2 : {R2, R3} implExactly {B, C}
Input 3 : {R4} implAtMost {C, D}
```

For the simple illustration discussed in Section 2 and the three inputs discussed previously and repeated above, Fig. 2 shows the corresponding footprint graph. The middle two columns depict the code elements (CE-nodes) for *A, B, C, D*, and *E*; and the artifact elements (AE-nodes) for *R*1, *R*2, *R*3, and *R*4. The left column depicts the artifact element groups (*AEG*) by connecting each set of artifact 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 artifact element. This graph structure depicts the certainties as connections between CE-nodes and AE-nodes and uncertainties as connections between CE-nodes and AEG-nodes or AE-nodes and CEG-nodes. In terms of scalability, the footprint graph structure grows linearly with the user input ($\#totalnodes = \#CE{-}nodes + \#AE{-}nodes$).

The footprint graph is the foundation for automatic trace generation. During trace generation, the artifact elements in the graph

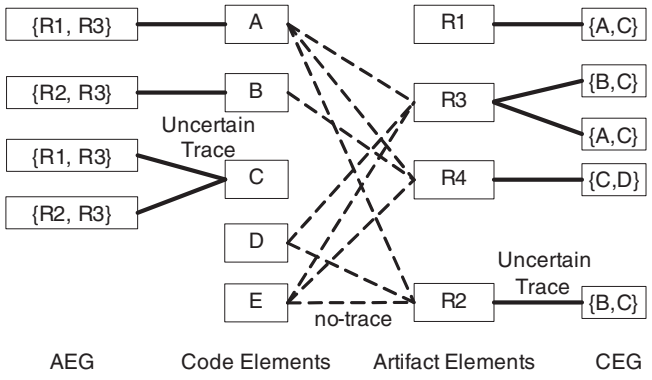


Fig. 2. Footprint Graph from Input.

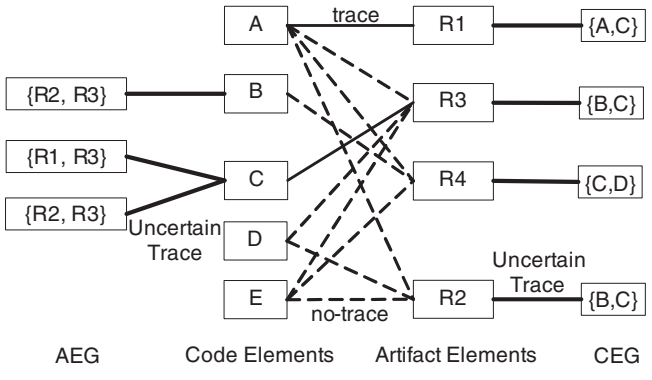


Fig. 3. Footprint Graph after Propagation Rules.

are propagated from the CEG-nodes and AEG-nodes (containing the uncertainties) to the CE-nodes and AE-nodes (connected by the certainties). There are several such propagation rules discussed below which are presented in our previous work (Ghabi and Egyed, 2012) for model-to-code traceability whereas (Egyed, 2004) supported only one of those rules.

4.4.6. Propagation rules for reducing uncertainty

Consider the example in Fig. 2 once again. The first input resulted in *aeg*(*A*, {*R*1, *R*3}) implying that *A* must implement either *R*1 and/or *R*3. Then the third input resulted in *no-trace*(*R*2, *A*) and *no-trace*(*R*3, *A*). So if A is supposed to be implementing {*R*1, *R*3} but *A* is not supposed to implement *R*3 then clearly *A* must be implementing *R*1 – the only remaining artifact element in the *AEG*. Recall that the *AEG* defines a constraint over multiple artifact elements where at least one of these artifact elements has to be implemented by the code element. Uncertainties in an *AEG* can thus be resolved (or reduced) by eliminating artifact elements that are known to not implement the code based on other input:

```
if no−trace (ae, ce)
  forall ceg: CEG where ce in ceg. {ce∗}
    ceg. {ce∗} := ceg. {ce∗}−ce
  forall aeg: AEG where ae in aeg.{ae∗}
    aeg.{ae∗} := aeg.{ae∗}−ae
```

4.4.7. Propagation rules for suggesting trace

Uncertainties in a *CEG* are resolved similarly. For example, the first input also resulted in *ceg*(*R*3, {*A, C*}) implying that *R*3 was supposed to be implemented in either *A* and/or *C*. Since *R*3 was excluded from code element *A*, the *CEG* is left with only one code element, namely *C*. This remaining code element must be implementing *R*3 for *CEG* to be satisfied.

```
if ceg. {ce∗}. size = 1 then
  trace (ceg. ae, ceg. {ce∗}. first)
  remove ceg
if (aeg. {ae∗}. size = 1) then
  trace (aeg. {ae}. first, aeg. ce)
  remove aeg
```

Fig. 3 shows the footprint graph after the application of the propagation rules discussed above. Note that the certainty increased as the links between *AE*s and *CE*s increased while uncertainty decreased because there are fewer *CEG* and *AEG* nodes now. The propagation rules are applied for as long as possible. The order in which the rules are applied is irrelevant.

4.4.8. Consistency constraints

Input given by the engineers may be partially/fully generated by hand and may be based on potentially outdated documentation or second-hand information (i.e., from previous project members). Consequently, the input given by the engineers cannot be fully trusted – indeed it may even be inconsistent where one engineer's understanding contradicts another engineer's understanding. Our approach assumes that information provided by the engineers is correct unless it violates correctness checks. Fortunately, not every input combination is valid and our approach identifies four forms of input inconsistencies. Do note that consistency does not imply correctness; however, with increasing quantity of input it becomes increasingly unlikely that an input containing errors remains consistent, especially if the input is provided by different engineers (we validate this in Section 7.2.2.1). The following demonstrates how our graph structure supports correctness checking.

(1) Every AEG *must have at least one artifact element:.*

$$\forall aeg \in AEG, \quad aeg.size > 0$$

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

Input 4: {R1} implNot {A}

Recall from Section 4.4.6 that the *aeg*(*A*, {*R*1, *R*3}) from input 1 was previously reduced to *trace*(*R*1, *A*) because *R*3 is not implemented by *A*. If another engineer now states that *R*1 is also not implemented by *A* then the *AEG* is left without an artifact 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:.*

$$\forall ceg \in CEG, \quad ceg.size > 0$$

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

Input 5: {playing} implNot{C}

Recall from Section 4.4.7 that the *ceg*(*R*3, {*A, C*}) from input 1 was previously reduced to *trace*(*R*3, *C*) because *R*3 was not implemented by *A*. If now *R*3 is also not implemented by *C* then the *CEG* is left without a code element.

(3) Every artifact element must be implemented by some code:. The premise of this approach is that it works only on artifacts that are implemented in code. Even if no *CEG* or *AEG* is violated, we must still make sure that every artifact element is implemented by some code. This check is particularly useful for those artifact elements in perspectives for which no input was defined.

(4) A code element cannot be implementing and not implementing an artifact element at the same time:. A CE-node contains the certainties of the input and the resolved uncertainties of the CEG-nodes and AEG-nodes. These certainties should not conflict such that a code element implements and not implements the same artifact element. Obviously, saying *R*2 is implemented by *A* and *R*2 is not implemented by *A*, produces this kind of error – though in practice such conflicts are hard to see manually if multiple input are involved.

4.4.9. Granularity constraints

While software development standards mandate the establishment of traces between artifact 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 of establishing traceability to Java classes, Java methods, or even individual lines of code. It is also possible to establish the traceability to Java packages or any other architectural grouping (e.g., client code vs. server code).

Obviously, the level of granularity vastly affects the cost of trace generation. In Egyed et al. (2005), we determined that the input quantity of the artifact-to-class mappings was almost 10 times less than the input quantity of the artifact-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 Egyed et al. (2005), we also discussed that a coarser granularity resulted in quality loss because functionality was grouped together that was separated on a finer level of 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). But in same study 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).

Obviously, what granularity rate to choose depends on the needs of the engineers and the effort they are willing to put in. Previously, we argued that the granularity should be staged depending on the importance of the artifact element. One may start off by defining the granularity on a coarser level (e.g., artifact to Java classes) and then refine key areas to a finer level of granularity (e.g., artifact to Java methods). Here we propose an additional avenue by defining granularity constraints that suggest which code elements to refine. This is discussed next.

(5) 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 {R1} implNot {A} caused a correctness violation because the code element excluded both artifact elements *R*1 and *R*3 from the *AEG*. But what if code element *A* was too coarse grained and should have been broken down into methods, say *A*1 and *A*2. The following input, on a finer level of granularity, resolves the conflict:

Input 1: {R1, R3}implAtLeast {A1, C}

Input 4: {R1} implNot {A2}

Correctness violations indicate problems where the input cannot be reconciled. Granularity 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 (we discuss later how this is done correctly automatically). A selected refinement of *A* only 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.

4.4.10. 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 artifact elements and/or code elements that are still incomplete. For an artifact element to be complete, it must have traces and *no-traces* to all code elements:

$$complete(ae) \Rightarrow$$
$$\#trace(ae) + \#no{-}trace(ae) = \#CE$$

The completeness of an artifact element can be determined for every artifact element separately. A code element implementing an artifact element '*ae*' is complete if all other artifact elements of the same perspective $AE^P - \{ae\}$ are either defined as trace or no-trace.

5. Encoding and correct reasoning

One of the main challenges is to correctly reason about the language presented above – even in the presence of inconsistencies. On the surface, we require a reasoning engine that allows us to encode the facts, uncertainties, and constraints. The reasoning engine's main responsibility would be to refine the input through the propagation of the rules discussed in the previous section and to identify constraint violations. We tested this on two, quite different reasoning engines: (1) Drools (DRO, 2014) – an incremental business reasoning engine; and (2) PicoSAT (Biere, 2008) – a light weight yet powerful SAT Solver. Both engines were capable of performing the required computations and delivered correct results though the performance of PicoSAT was far superior to Drools. But below the surface, we also require a reasoning engine that functions correctly in the presence of inconsistencies – a key requirement since inconsistencies among different engineers' assumptions are expected to be the norm and not the exception (Egyed et al., 2010). Here the SAT reasoning engine was superior because we could augment its reasoning to support correct reasoning in the presence of inconsistent input. The following discusses both implementations and also how correct reasoning is possible through correct isolation of inconsistencies.

5.1. Encoding traceability language in drools

Drools is a Business Rules Management System developed by JBoss. It delivers a very powerful rules engine (Drools Expert) with its own rules language (DSL). Each rule has a condition. Each time the condition is satisfied, the action of the rule will be executed. The rule depicted in Listing 1 shows how *AEG*s and *CEG*s are generated from *implAtLeast* constructs. Further rules handle the different types of constructs and all the reasoning of reducing uncertainty from *AEG*s and *CEG*s.

The rules language is straight forward and could easily be extended to cover new rules (see future work). This is a key feature of our tool TraceAnalyzer which will be introduced later (see Section 6). Unfortunately, the Drools reasoning engine (Version 5.x) is based on backward chaining reasoning. Once the reasoning engine is started, the engine fires the rules in the knowledge base and then checks if

Listing 1. Derive AEG/CEG from ImplAtLeast with Drools

```
rule "Generate facts from 'ImpAtLeast'"
when
  $c : Construct( type == IMPL_AT_LEAST )
then
  for (Artifact src : $c.getSources()) {
    createAEG($d, src, $d.getTargets()) );
  }
  for (Artifact tar : $d.getTargets()) {
    createCEG($d, tar, $d.getSources()));
  }
end
```

there are conditions satisfying the fired rules. If the condition is not satisfied the result from the fired rule will be ignored as if it was not fired. This kind of reasoning is inefficient as it computes unneeded facts and waits to invalidate them later when the condition of the rule is not satisfied. E.g. the rule in Listing 1 will be called for every construct with type *implAtLeast*. The created *AEG*s and *CEG*s will be effectively added to the knowledge base only for input construct of type *implAtLeast* and they will be ignored for all other constructs. The Drools engine performs very well with a moderate number of input constructs but its scalability suffers given the exploding number of rules fired even if they are not satisfied. During our evaluation we faced many cases where the engine required hours to resolve the input. Therefore we developed another implementation of our approach using a SAT Solver. This reduced the runtime of some case study systems from hours to seconds.

5.2. Encoding traceability language in SAT

SAT solvers are highly efficient tools designed to check the satisfiability of a problem encoded in conjunctive normal form (CNF). Encoding a TM of a given system into a CNF input is a straightforward operation. Each cell in a TM is basically a Boolean literal that could be "true" if the cell contains a trace, or "false" if the cell contains a *no-trace*. Depending on the uncertainty input, our approach derives a number of possible *traces* and *no-traces* that fill up some cells in the TM. We define the values of the literals corresponding to the filled cells by their *trace*/*no-trace* value. The empty cells are encoded as variables (undefined). Every *AEG* or *CEG* resulting from the input is translated into a clause – a disjunction of all the literals referenced by the corresponding *AEG* or *CEG*. The conjunction of all the clauses resulting from encoding all the *AEG*s and *CEG*s builds a CNF formula, which is the presentation of the initial uncertainty input that we use as an input for the SAT solver.

As an example, consider the construct Input 1: {R1, R3} implAtLeast {A, C}, which can be decomposed into the *ceg*(*R*4, {*C, D*}) and several *no-trace* relationships such as *no-trace*(*R*4, *A*). Each cell in the TM is represented by a literal in the CNF formula, in this case by the literal *R*4−*A*, with the semantic of being positive for *trace* and negative for *no-trace*. Since each *no-trace* relation corresponds to one cell exactly, they can be directly translated to clauses containing a negated variable. For example, ¬*R*4−*A* says that *R*4 does not trace to code element *A*. The semantic of the *ceg*(*R*4, {*C, D*}) is that *R*4 is implemented by *C* or *D* or both of them. As a consequence, the cells *R*4−*C* and *R*4−*D* cannot both be *no-traces*. This can be represented as the clause *R*4−*C* ∨ *R*4−*D*. However, since both the *no-trace* and the *CEG* relationships stem from the same construct provided by an engineer, we add a clause selector variable to the clauses stemming from a single input construct, i.e., *input*1. As a result, for our example the construct {R4} implAtMost {C, D} is translated to the following CNF:

$$(\neg input1 \vee R4{-}C \vee R4{-}D) \wedge (\neg input1 \vee \neg R4{-}A)$$

So, if the clause selector variable *input*1 is true then the clauses are included into reasoning, otherwise they are ignored (this is important for correct reasoning discussed below where we need to isolate offending inputs). For the sake of brevity we have shown an example of selectors on construct level, but we should note that the granularity of our reasoning could be arbitrarily changed by changing the reference of the selector variables: e.g. we can introduce selector variables to reference each clause of *CEG*s or *AEG*s, or even literals of cell in the RTM. The finer the granularity of the selector variable, the more selectors are added to the CNF. A finer granularity makes the CNF bigger, which requires the SAT engine to spend more reasoning time. It is a tradeoff between performance and the granularity of the results.

5.3. Reasoning about traceability in SAT

The SAT solver checks whether the input CNF is satisfiable (SAT) or unsatisfiable (UNSAT). A satisfiable CNF input means that there is at least one set of assignments for all the literals allowing the entire input CNF to evaluate to true. Considering the assumptions provided in the CNF, its satisfiability means that the encoded problem does not contain any inconsistencies (though it does not necessarily mean that it is correct). Furthermore, the PicoSAT solver allows for an efficient oracle to investigate the assignments of the remaining literals, filling up the remaining cells in the TM. For example, we already know that *R*4−*A* must be a *no-trace* because it was defined to be false above. But, what about *R*4−*C* or *R*4−*D*? Presently both could be either true or false individually but not both. Yet in the presence of additional input, this conclusion may no longer be valid and we use the SAT solver as an oracle to automatically test whether these assignment are valid by adding yet another clause, one at the time. For example, we would add *CNF* ∧ (¬*R*4−*C*), then separately *CNF* ∧ (*R*4−*C*) to test whether the CNF is still satisfiable with and without *R*4−*C*. If *CNF* ∧ (*R*4−*C*) was satisfiable, but *CNF* ∧ (¬*R*4−*C*) was no longer satisfiable then we may conclude that (*R*4−*C*) must be true and hence a *trace* – a logical conclusion of the input provided. The entire procedure of adding a clause and testing it is done automatically. The UNSAT state of CNF thus has two interpretations: (1) UNSAT on the input CNF implies that the input is inconsistent and (2) UNSAT on the oracle CNF implies that the added clause is no longer feasible. The problematic case is thus the first one which is discussed next.

5.4. Correct SAT reasoning with Humus

SAT solvers fail if the input CNF is not satisfiable. This is a major problem here because we expect the different inputs from different engineers to be inconsistent as a norm and not as an exception. The question we answer next is how to enable SAT-based reasoning in the presence of inconsistencies. Nöhrer et al. (2012) compared different isolation strategies for dealing with such inconsistencies. The most obvious one is MAXSat (Li and Manyà, 2009), which identifies a maximal subset of the CNF that is still satisfiable. In other words, MAXSat isolates (i.e., temporarily discards) as many clauses as needed to make the CNF satisfiable again. The problem that Nöhrer et al. also demonstrated was that such isolation does not ensure correct reasoning thereafter. The reasons are this: for an inconsistency there must be at least two contradictory clauses. For example, let us assume another input that says that *A* traces to *R*4, which adds (¬*input*2 ∨ *R*4−*A*) to our CNF:

$$(\neg input1 \vee R4{-}C \vee R4{-}D) \wedge (\neg input1 \vee \neg R4{-}A) \wedge (\neg input2 \vee R4{-}A)$$

Clearly, the second and third clauses are contradictory – a trivial observation here. Either the second clause is wrong or the third one is (or perhaps even both). By isolating any one of the two clauses, the inconsistency is eliminated and the CNF becomes satisfiable. MAXSat cannot know which cause is correct and which one is wrong and it simply makes a random decision that emphasizes the minimum. It may thus isolate the correct clause (if there is one), leaving the

wrong clause in the CNF. The CNF becomes satisfiable but the reasoning about traceability in SAT (Section 5.3) would now be affected by the still present, wrong clause – *hence, the reasoning would no longer be correct*.

Addressing the problem of MAXSat, Nöhrer et al. (2012) introduced the High Level Union of Minimal Unsatisfiable Sets (HUMUS) strategy which computes the union of all Minimal Unsatisfable Sets (MUS) responsible for the CNF to be unsatisfiable. The basic concept behind it is to isolate all clauses (directly and indirectly) of the inconsistency and only keep clauses that have no relation to the inconsistency. Continuing on the example above, HUMUS would isolate both clauses – $(\neg input1 \vee \neg R4\text{–}A)$ and $(\neg input2 \vee R4\text{–}A)$ – thus assuring that the wrong clause is definitely isolated even though at the expense of also isolating the correct clause (if there is one). Consequently, the HUMUS isolation ensures correct reasoning about traceability in SAT after isolation – though at the expense of incomplete reasoning: since HUMUS would isolate both clauses it would also isolate the other, presumed correct clause which reduces the CNF and leads to less complete reasoning. In the most extreme case, HUMUS would isolate everything if all clauses where directly or indirectly involved in the inconsistency. However, in practice this is not the case and HUMUS is in fact quite efficient in its isolation – the incompleteness it causes is small and its effect quickly dissipates which we will discuss further in the evaluation.

The exact functioning of HUMUS is discussed in Nöhrer et al. (2012) but it should suffice to say that HUMUS computes the maximal satisfiable set (MSS) which is the opposite of the MUS. The MSS is the maximal subset of assumptions such that adding any further assumption would turn a satisfiable CNF into unsatisfiable. On the encoding level, HUMUS requires the introduction of additional variables (literals) in the input CNF, the so called selector variables we introduced earlier. They are used to link each clause to its corresponding input such that all clauses of an input are isolated at once. For example, since $(\neg input1 \vee \neg R4\text{–}A)$ is involved in the inconsistency which stems from *input*1, we might also not want to trust the other clause from *input*1, namely: $(\neg input1 \vee R4\text{–}C \vee R4\text{–}D)$. Though not a direct contributor, it is indirectly related to a clause that needs isolation. The existence of those selectors do not influence the result of the SAT reasoning, therefore they are always true prior to inconsistencies. As HUMUS identifies clauses involved in inconsistencies, these clauses can then be efficiently isolated by changing their selector variables to false.

6. Proof of concept tool

Our approach is fully implemented in a tool called TraceAnalyzer (see screenshot in Fig. 4). This tool is implemented on the Eclipse platform to offer a familiar user interface for engineers. It allows for different input views: from the traditional trace matrix (TM) shown right to the list of inputs shown left. It also lets the engineer investigate the footprint graph (shown at the back), highlights correctness and granularity problems, and isolates them if desired. The TraceAnalyzer is built in a modular structure (common eclipse plug-in structure) that allows different types of reasoning engines to be used including the business rule engine Drools (DRO, 2014) and the SAT solver PicoSAT (Biere, 2008). Each of these reasoning engines could be easily extended by implementing the extension points defined for traceability reasoning. The extension of Drools engine is as simple as adding new rules to the existing knowledge base (see example in Listing 1). The extension of SAT solver requires more elaboration as an adaptation of the CNF mapping is needed. Though presently the PicoSAT reasoning engine is superior as was discussed. The validation next also focuses on this reasoning engine.

It is also noteworthy that TraceAnalyzer supports different granularity levels: construct level, *CEG*/*AEG* level, and RTM cell level. The engineer could change the granularity level in the standard eclipse settings. This allowed us to verify our approach on the different levels and assess its performance in best case (construct level) and worst case (cell level) scenarios.

Table 1
Useability experiment case study systems.

	CMA	Design space	Ecco
Language	Java	Java and C#	Java
KLOC	5.6	91	33
Code Elements	158	80	251
Requirements	8	29	33
Size of RTM	1264	2320	8283
Per Subjects			
Subjects	2	3	3
Familiarity	40 – 95%	70 – 95%	50 – 95%
Uncertainty (i)	100%	65 – 100%	75 – 90%
TraceAnalyzer			
T/N Conflicts	38	453	832
Inconsistencies	28	54	192
Uncertainty (ii)	62.5%	0%	33%

7. Validation

Our approach deviates strongly from conventional trace capture approaches and introduces a new paradigm for creating traces. With our approach, the engineer is allowed to describe his/her knowledge about traceability between artifacts and code without being bound to providing complete information or even correct information. The evaluation thus focuses on four key aspects:

- Usefulness: Do trace uncertainties and inconsistencies exist in practice and does our approach reduce them?
- Correctness : Does our approach reason correctly - even in the presence of uncertain and erroneous trace input.
- Completeness : Does the isolation of erroneous input affect the completeness of the reasoning (because correct input will be isolated also)?
- Scalability: Does our approach scale to large traceability problems (quadratic growth).

To investigate these aspects, we rely on six case study systems: three for usefulness and the other three for correctness, completeness, and scalability. Separate case studies were needed for the usability study because of its different prerequisites. The usability study required immediate access to subjects and case studies to use our language. On the other hand, the correctness, completeness, and scalability studies required a gold standard of correct traces to compare against.

7.1. Usefulness

To assess usefulness, our approach presumes that engineers find it easier and more intuitive to capture traceability using a language that has explicit mechanisms for handling uncertainty - mainly because we presume that each person has an incomplete and presumably even inconsistent perspective of the traceability of a given system. To test this basic assumption, we performed a controlled experiment involving eight subjects and three case study systems (listed in Table 1). CMA is the configuration management module in an industrial product implemented in Java. It has eight high level functional requirements implemented in 158 classes (code elements). Performing conventional traceability on such a system required engineers to fill in a traceability matrix with 1264 cells (8 requirements multiplied by 158 code elements). Two software engineers (both employed by the company building CMA) volunteered to capture this traceability using our approach. Furthermore, DesignSpace and Ecco

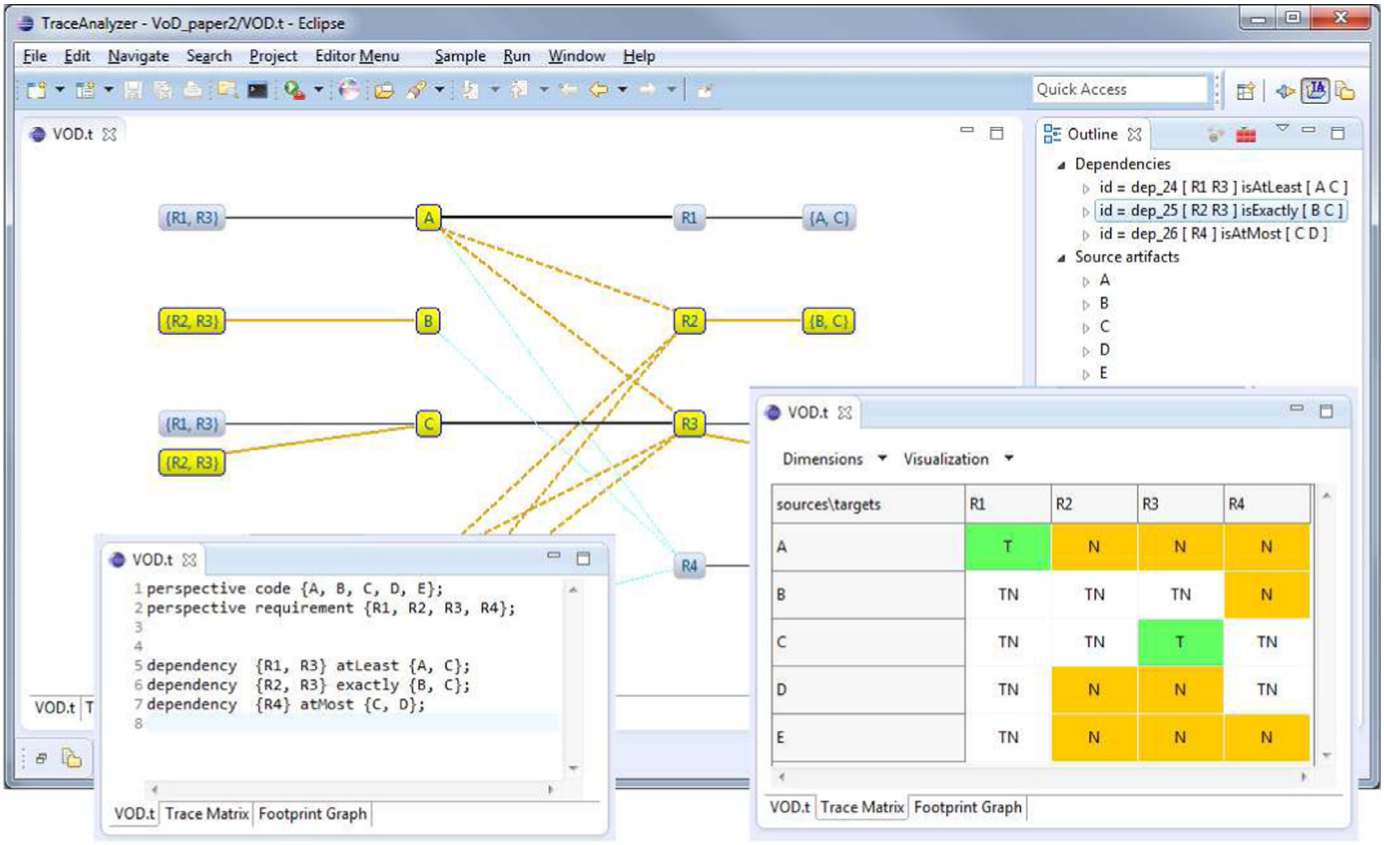


Fig. 4. TraceAnalyzer Screenshots.

are larger research projects developed at the Johannes Kepler University (JKU). There, three graduate students volunteered to capture the traceability for each system where 29–33 requirements were traced to 80–251 code elements. The three systems ranged from 5–91 KLOC in size.

The experiment was structured such that we had at least two subjects per system in order to be able to identify the existence of (a) conflicts and (b) the complementary effects of traceability uncertainties - the basic premise of our approach. Note that uncertainty is an individual problem where a subject is unable to identify traceability precisely. Should uncertainty indeed be a major problem then we ought to observe this among the eight subjects and the 70 requirements they investigated. Conflicts and the complementary effects of uncertainties reveal themselves only if multiple subjects capture traceability independently of one another. If they do exist then we ought to observe them among the 11,867 cells that make up the three systems' RTMs.

We devised a controlled experiment whereby the eight subjects were asked to recover the traceability of the three case study systems. The experiment was preceded by a training session of about one hour to explain the goal of traceability and to teach the uncertainty constructs. We answered the questions raised by the participants without biasing their judgment. We encouraged them to be precise and complete in their assessment but to refrain from guessing answers in case of uncertainty. We also suggested alternative forms of capturing traces and even allowed them to come up with their own constructs if they deem necessary.

After the training session, the experiment started and no further support was provided. First, the subjects of each case study system had to work together to identify the requirements and code elements they would use for the study. In doing so, they identified 8–33 requirements and 80–251 code elements (depending on the case study). Each developer then captured the traceability between those requirements and code elements separately. Finally, we collected the traceabilty the subjects captured and analyzed them.

Table 2
Correctness, completeness, and scalability case study systems.

	ReactOS	Gantt	JHD
Language	C++	Java	Java
KLOC	18	41	72
# Code elements	239 (C)	2591 (M)	1763 (M)
# Requirements	16	18	21
Size of gold standard	3824	46638	37023

We observed that uncertainty was prevalent throughout all three systems, all eight subjects, and most requirements. Indeed, we found that each subject had a slightly different understanding of the system and how it was implemented. Table 2 lists that 65–100% of the requirements captured by the subjects had some uncertainty (i) - that is the subjects usually could not tell with certainty all the code elements that traced to a given requirement even though all of them war highly familiar with the system overall (see Familiarity). This strongly confirms the existence of uncertainty and thus motivates our approach.

We also analyzed whether subjects contradicted and/or complemented each other in their traceability knowledge. The easiest example of a conflict is a direct trace/no-trace conflict (T/N conflict) where one subject claims a trace between a given requirement and a code element while another subject does not. Our approach detected between 38 and 832 such trivial conflicts. However, more important are the existence of non-trivial inconsistencies (recall 4.4.8). We identified between 28 and 192 such inconsistencies. This strongly confirms the need for the error detection mechanism of our approach.

Furthermore, we analyzed whether requirements with uncertainties got reduced when combining uncertain input from multiple developers (recall 4.4.6 and 4.4.7). This is indicated in the Table as uncertainty (ii) and we see that the uncertainty after applying our approach was reduced strongly compared to the individual uncertainties that existed beforehand. For example, our approach reduced the 65–100% uncertainty in the DesignSpace to 0% uncertainty, which means that all uncertainties were resolved for all requirements. This confirms the benefits of our approach.

After the experiment a short free-form discussion took place with each subject individually. Our intent was to ask the subjects whether they had any traceability knowledge which they could not express with the constructs provided by our approach. In response to this question, two developers suggested bi-directional language constructs. In the current language, it was only possible to define situations like: requirement1, requirement2 are at most implemented by code1, code2, code3 but those two subjects also desired to express reverse situations like: code1, code2 are at most implementing requirement1, requirement2. This is a legitimate use case that we intend to address in future work (see Section 9). Furthermore, two subjects reported problems in defining an appropriate level of granularity (recall 4.4.9). For the most part, the subjects traced requirements to Java packages or classes. However, the subjects felt the need to refine the traceability for certain classes to method to better distinguish their traceability. This refinement process was not supported and this is also an interesting use case for future work.

7.2. Correctness, completeness, and scalability

Next, we discuss our approach's correctness and scalability. We consider two kinds of input: (1) correct input, which presents a consistent knowledge about a system without any error involved; and (2) input with incorrectness containing errors. Correct input should not trigger inconsistencies. Incorrect input may or may not trigger inconsistencies and hence may or may not need isolation. For as long as it is not detected as an inconsistency, it negatively affects the reasoning, which explains the need to understanding the likelihoods that such incorrect input remains hidden. But once it is detected, the HUMUS isolation is guaranteed to isolate it (the proof for this can be found in Nöhrer et al. (2012)). Yet, the isolation may also affect correct input which reduces the completeness of the reasoning. Hence, the need for investigating the completeness. In the remainder of this paper we refer to detected errors as meaning that an inconsistency on cell level was found.

For understanding correctness, completeness, and scalability, we draw on another set of case study systems because we require a gold standard of correct and complete traceability to assess our approach. For this, we identified three case study systems: Gantt (GAN, 2014), JHotDraw (JHO, 2014), and ReactOS Explorer (REA, 2014). All three systems are described in Table 2. Gantt and JHotDraw are open source programs implemented entirely in Java. The ReactOS Explorer is the file explorer implemented for ReactOS which is an open source operating system compatible with the Microsoft Windows-NT architecture and implemented in an object oriented C++ paradigm. We chose these systems because we had available high quality requirement trace matrices (RTM) that contained all the traces between requirements and methods/classes. Each RTM was captured by a key developer of the system involved which guarantees the best quality possible. These RTMs are our *gold standard* which were attentively created by key developers of the different systems and we believe that they have the best possible quality for a conventional traceability. We use them to assess the results computed by our approach.

The empirical evaluation in Section 7.1 has proven the usefulness of our approach on three case study systems (see Table 1). The focus of the remaining evaluation is thus on correctness, completeness, and scalability. To explore this, we systematically investigated all possible uses of our approach by simulating virtual engineers using our technique on the aforementioned industrial projects (see Table 2). The intent of the simulation was to demonstrate that the approach is applicable, functions correctly, and scales under many different kinds of usage scenarios in context of real projects. For each of the case study systems we generated input constructs (*implAtLeast, implAtMost …*) based on the correct TMs we had available for each system. By systematically generating different combinations of such input constructs we simulated different uses of our technology. This form of evaluation is useful to adequately answer the research questions laid out above (i.e,. scalability, correctness, likelihoods, or completeness) as we essentially explore all possible uses. What we cannot show through this evaluation is the usefulness of our language (i.e., would engineers prefer to use it over traditional TMs?). However, the intuitiveness of our language compared to the unnecessary strict TMs suggests that the language should be useful which is the focus of further user studies.

Focusing on the key aspects above, we thus generated a large number of input sets. Each input set had 'n' input relationships which consisted of arbitrary ratios of each construct: e.g. we had an input with 30 relationships that included 10% implNot, 20% implExactly, 30% implAtLeast, 40% implAtMost; or we had input with 100% implExactly. The tricky part of this simulation does not reside in generating the input itself, but rather systematically varying the ratios in order to cover all possible usage scenarios. We believe that different engineers at different states in development are likely to use different ratios of our language constructs. E.g., a knowledgeable engineer is more likely to use implExactly than implAtMost. Some engineer working on details may not know other parts and may thus be more likely to use implAtLeast. We thus generated inputs with variations of: (1) number 'n' of constructs, and (2) the ratio 'r' of each construct type in the input. Generating all possibilities results a large number of inputs and we thus used an automated engine to explore them. For test purposes, the number of constructs 'n' was varied between 30 and 100, and the distribution of ratio 'r' will jump by 25%, either by adding 25% more or less of a given construct type.

Additionally to the variations discussed above, we also studied the effect of errors on the correctness of the results delivered by our technique. There is no guarantee that the engineers would have a correct understanding of a system. For this purpose we use the same inputs generated by the variation of 'n' and 'r' as discussed previously and seeded errors. Like the case study systems, the seeded errors were not random but based on actual errors we observed with subjects performing trace capture (Egyed et al., 2010). Our evaluation thus systematically explores all possible uses of our technology on three real case studies involving real trace errors observed in practice.

7.2.1. Evaluation with correct input

A correct input is a set of uncertainty constructs describing correct (though not necessarily complete) traceability knowledge about a system. In total we generate about 200 different inputs without seeded errors for each of the three case study systems in the manner discussed above.

7.2.1.1. Completeness of uncertainties. To evaluate the completeness of our approach, we investigated the degree of traces/no-traces gained from analyzing uncertainty constructs by the number of RTM that could be computed compared to the original gold standard RTM. We refer to this number as cell coverage (the higher, the more complete). Fig. 5 shows the percentages of cell coverage relatively to the gold standard RTM of each case study system while increasing the input size. We measure the input size by the number of *uncertainty groups* UGs (the set of AEGs and CEGs together because the UG size corresponds directly to number of clauses in SAT) derived from the input constructs. We notice the curved nature of the diagrams. It

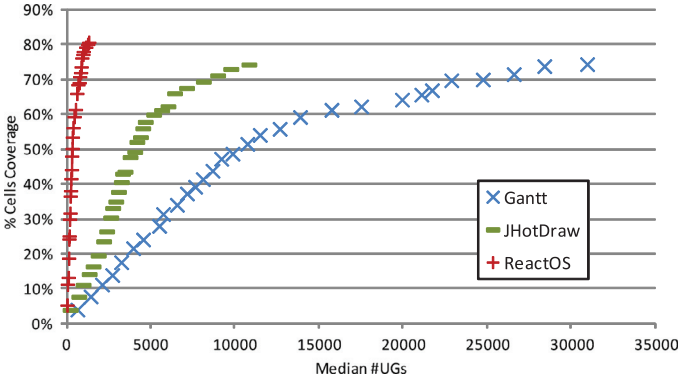


Fig. 5. Completeness with correct Input.

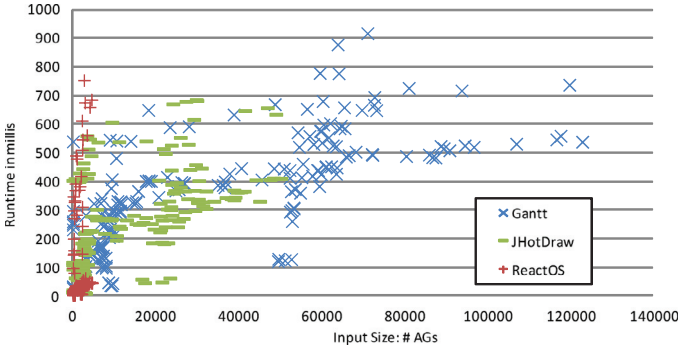


Fig. 6. Scalability with correct Input.

shows that the cells coverage increases quickly with few uncertainty constructs. This implies that uncertain input allows us to derive certain facts quickly and this effect is stronger in the beginning and slowly reduces. We thus believe that our approach is most useful for quickly gaining good coverage when brainstorming among engineers. But as the curve implies, it would be hard to obtain a 100% coverage through such "guessing". At a certain level of coverage, it is likely necessary to explore the missing cells and refine them. The completeness constraints generated by our approach would then be useful to guide the engineer in resolving the remaining uncertainties.

Fig. 5 presents the mean values of all test inputs generated for each test case system relatively to the number of UGs. The actual cell coverage percentage varied depending on the type of constructs used in the input but all exhibited this effect in principle.

7.2.1.2. Correctness. A correct input consists of a set of uncertainty constructs which are consistent (do not contradict with each other) and do not contain any wrong assumption about the system which they are describing. From such a correct input our approach is expected to only derive correct *traces*/*no-traces* but no inconsistencies. We thus compared our approach's results with the gold standard to verify this. In total more than 33 million *traces*/*no-traces* relationships were generated from the 597 inputs and none of them caused any inconsistencies (i.e., no false inconsistencies). Furthermore, all inputs were correctly refined by our approach and the generated *traces*/*no-traces* were indeed a subset of the *traces*/*no-traces* available in the original gold RTM (subset because none of the n input allowed for a complete generation of the gold matrix). The very large number of input and *trace*/*no-trace* comparisons with the gold standard does not guarantee correctness but strongly supports our claim that the approach functions correctly in the presence of correct input (incorrect input is discussed below).

7.2.1.3. Scalability. The performance and scalability of a system is best measured by the throughput (size of the handled problem) and the time required for the operation. Fig. 6 shows the performance of TraceAnalyzer on test inputs generated for each of our test case systems (Gantt, JHotDraw, and ReactOS Explorer). The input size was again measured by the number of uncertainty groups (UGs) at the *x*-axis and the runtime in milliseconds at the *y*-axis. For the sake of brevity we limited the displayed data to test inputs with highest number of UGs which are most meaningful for performance and scalability measurement. Each data point in the diagram presents the time (milliseconds in the *y*-axis) consumed by our tool to analyze a set of constructs (input) with the corresponding problem size (number of UGs in the *x*-axis). The time appears to grow strongly initially with few UGs and then flats off quickly with increasing UGs (larger SAT problems). We believe this is a reflection of the fact that little input allows for many possible interpretations which are computationally expensive to explore whereas more input implies more certainty and hence is faster to explore. Some outliers are visible in the figure but they are very few and they do not deviate very much from the general performance observed. The time required by the TraceAnalyzer to analyze and derive all the possible *traces*/*no-traces* is always under one second even with inputs containing more than 100.000 UGs, which shows that our approach is very scalable on correct inputs. Such a short reasoning time allows us to get a very fast response time in TraceAnalyzer, and thus we get an instant feedback about the set of uncertainty constructs that we are writing into the tool.

It is important to note that our approach is incremental: i.e. adding or retracting a construct to/from an analyzed input would not trigger a complete new analysis from scratch because TraceAnalyzer will add/retract the needed information for reasoning to/from the existing information from previous reasoning. This significantly reduces the runtime performance of the reasoning engine (incremental reasoning).

While we appear to observe a linear behavior of the execution time based on batch evaluations, we found that our approach performs much faster if relationships are added incrementally (data omitted for brevity). The incremental nature of our approach introduces more flexibility for the engineer. As a main benefit, we name the fact that the engineer is not obliged to run the entire analysis after each change s/he makes. TraceAnalyzer will detect the changes and rerun the analysis incrementally for the changed input only.

7.2.2. Evaluation with partially incorrect input

As was explained above, an erroneous input is likely in traceability. An incorrect input about a system should lead to an inconsistency which is reported back to the engineer once detected. Every inconsistency should be tracked back to its origins and the exact constructs responsible for it should be determined. Correctly understanding the incorrectness is not only important for the reporting to the user but also for isolating its effect to ensure correct reasoning. The engineer may fix the inconsistency right away if she/he has the correct knowledge. Or the engineer could let our approach isolate the inconsistency and continue reasoning about the rest of the input.

Inconsistencies in the input imply incorrect traceability knowledge. Consistent input does not necessarily imply correctness. Our approach is vulnerable to incorrect but consistent input (as are probably most software engineering techniques today). If the engineer provides an incorrect, but consistent input, our approach would not be able to detect any inconsistencies and it would generate traces/no-traces based on that incorrect input. However we find that trace capture is generally a task that multiple engineers have to perform and herein lies the strengths of our approach. Incorrect but consistent input is increasingly unlikely with increasing number of engineers involved. As a result, we expect input inconsistencies to be likely in larger projects and our approach will detect such inconsistencies and notify the responsible engineers. There is unfortunately

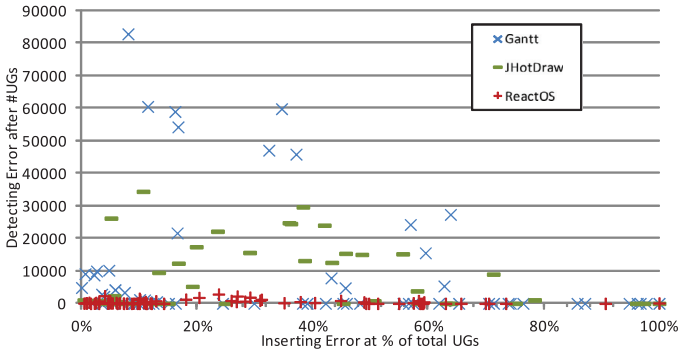


Fig. 7. Input Size to Detecting Errors.

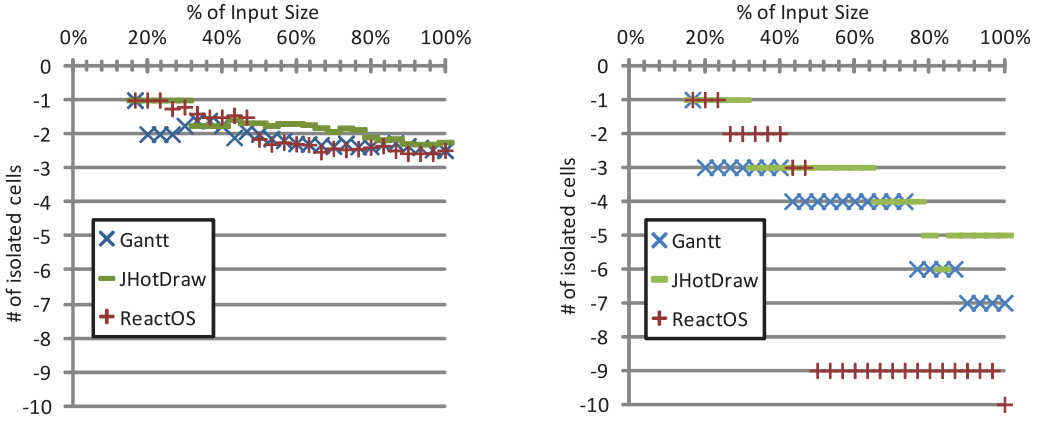
no study about how these errors are distributed or in which kind of relationship are they more likely. However, we previously did perform a controlled experiment where we let engineers recover traces for our three case study systems and (Egyed et al., 2010) and we thus had available a large set of data on real artifact to code traces that contained many errors (when compared to the gold RTM). We thus injected those erroneous traces into the otherwise correct inputs we used in the above validation – in total we generated 354 inputs with errors.

7.2.2.1. Likelihood of error detection. Not surprisingly, we found that the more input is provided the more likely an inconsistency is detected. We observed this behavior by computing the Detection Delay¥ between injecting an error and detecting it. Fig. 7 summarizes our findings. The *x*-axis depicts the size of the input (percentage of UGs from the their total number for each input) when injecting an error and the *y*-axis depicts how many more UGs were needed until the inconsistency was detected. Obviously the errors which are injected at the beginning, when the input is still small, stay undetected for a while until a considerable number of UGs is added. But when the error is injected at a point where many UGs already existed then it gets detected very quickly. This can be seen at the right of Fig. 7 where the number of UGs required until detecting an error becomes almost equal to zero. This observation confirms our previous argument that with increasing quantity of input it becomes increasingly unlikely that an input containing errors remains consistent, especially if the input is provided by different engineers.

7.2.2.2. Completeness of reasoning in the presence of erroneous input. Being able to detect inconsistencies is important for correct reasoning in the presence of inconsistencies. Recall that our approach is able to do so once the inconsistency is encountered – in which case, the isolation will eliminate the inconsistency conservatively by eliminating all clauses that contributed to it and thereby any/all errors. Previously, in Fig. 5 we showed that the amount of *traces/no-traces* covered rises very quickly at the beginning of the input and it flattens with increasing input. It stands to reason that by removing some input, the effect should be little compared to the size of the totally computed traces. Fig. 8 shows the difference of the coverage of the RTM cells gained from analyzing test inputs containing incorrectness compared to correct input. Every scatter in the diagram shows the average number of RTM cells (over all test inputs for each system) compared to the original covered cells when the input did not contain any errors. The percentage of the input size designates the number of constructs which are already analyzed by our tool. Fig. 8a shows the median number of cells lost in all our test inputs. The Fig. 8b on the right shows the maximum number of cells isolated in all our test inputs. The negative number of cells means that the input containing incorrectness covered less cells in the RTM than the correct input. We see that this number is very small and cover typically 10 cells or less. Compared to the total number of cells, the isolation thus affects <1%.

7.2.2.3. Correctness of reasoning with erroneous input. We argued that the HUMUS isolation is comprehensive. Thus, after the isolation, the approach is supposed to yield a less complete but correct RTM. The previous section demonstrated that the effect on completeness is very small. To test the correctness, we seeded single errors in inputs such that a detected inconsistency and isolation should eliminate the error always. We then compared the RTM after the isolation with the gold RTM to test the correctness of the reasoning and found that our approach was always correct after isolation. In the cases were our approach was not able to detect an inconsistency; we obviously could not detect the injected error. There was little benefit in measuring this error as virtually all software engineering approaches are susceptible to problems of undetected, incorrect input. However, as we already discussed above, the likelihood of an error remaining undetected decreases over time.

7.2.2.4. Scalability. The input with incorrectness requires more computational time because of the additional HUMUS computation to isolate the inconsistency. When HUMUS detects an inconsistency in the input, it does isolate the responsible input constructs. At any point of the reasoning, the isolation means two tasks: (1) all the reasoning parts, which are already computed based on those inconsistent constructs, should be rolled back; and (2) no further reasoning is allowed to take those inconsistent constructs into account. Fig. 9 shows the



(a) Median Isolated Cells **(b)** MaximumIsolatedCells

Fig. 8. Completeness with Incorrect Input.

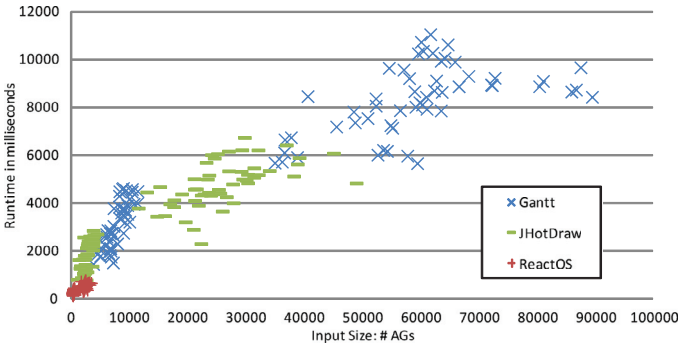


Fig. 9. Scalability with Incorrect Input.

time required to run the erroneous inputs. The maximum time required has increased from less than 1 s (see Fig. 6) to more than 10 s. Compared to correct test inputs, the total time needed for reasoning increases significantly, but we believe that this amount of time is still acceptable because it is a onetime cost. After isolation, the input is consistent again and the subsequent cost is the same as the reasoning without error (Section 7.2.1.3). The incremental nature of our approach makes it highly scalable even though the total time consumed to analyze an entire input has increased. Our approach requires more time to reason about uncertainty constructs which are in contradiction with other constructs. But in practice the engineer will feed the tool incrementally with single constructs and only when she/he adds an inconsistency then the tool would take few seconds to identify, isolate, and return a feedback, which is still an acceptable user experience. Again, we note that the computational time increases sharply for few UGs and flattens with increasing UGs. We thus do not expect that the computational cost will increase much further with larger case studies.

8. Threats to validity

Our approach is able to detect incorrect input. However, here a thread to validity is our assumption that incorrect input coincides with contradictory input. This must not be case always. For example, if two engineers have a wrong but consistent understand of the traceability of the system then out approach would not identify inconsistencies. Our approach is thus vulnerable to this problem which would limit its usefulness despite its rigor. Our approach is intuitive and caters to obvious uncertainties that an engineer would really have. However, here again we learned from the aforementioned controlled experiments (Egyed et al., 2010). While there were few hotspots where engineers tend to err consistently, most errors where isolated and engineers often had contradictory understandings (i.e., whenever multiple engineers investigated the same cell of a trace matrix then there typically where inconsistent opinions). This strongly supports our argument that incorrect but consistent input is possible but not common.

Another thread to validity is that we did not consider the cost of capturing traceability. Traceability is useful only if the cost of capturing traceability is less than the benefits of using it. However, this cost/benefit question is not specific about our approach but rather about traceability as a whole. Unfortunately, today we have no effective benchmarks to assess cost/benefit trade-offs in traceability because the uses of traceability are diverse and no studies exist that cover the cost. However, the perceived benefits are strong enough for existing standards (CMM, FAA) to mandate traceability. Our approach does not change the basic complexity of traceability (n^2) and it was not meant to be a mechanism for saving cost. The key question is thus whether our approach makes trace capture more complex than simply filling in a trace matrix. Here we see no reason that this should be the case either. After all, there is also the cost of wrong traceability to consider. If engineers are not able to separate precise from uncertain knowledge then engineers may fail to capture some traceability (incompleteness) or make arbitrary decisions with others (incorrectness). Neither is beneficial and our approach at least avoids those and their subsequent problems. So while the approach may not be a cost benefit itself, it is not a loss either and the more accurate reflection of traceability should benefit downstream uses.

The introduction also motivated the role of evolution in keeping traces complete and correct. Yet we did not consider evolutionary change to traceability explicitly. This is true but we believe that evolutionary changes are simply input changes and can be handled like any new/changed input. However, evolutionary changes have the risk of not having been applied consistently and, in such cases, our approach should detect inconsistencies. Hence, the evaluation with partially incorrect input was also an evaluation of evolutionary changes carried out inconsistently. Here the incorrect input would refer to older, outdated input that is now inconsistent with newer, evolved input.

Finally, the evaluation in this work focused entirely on requirements to code traces and not arbitrary artifacts. Yet, the problem of filling in a trace matrix is the same regardless of artifact type and we see no reason why the results ought to differ from, say, models. Again, merely the usefulness could be different though for assessing that we would need more user studies as was already discussed. On a smaller scale, we did validate models as well (Ghabi and Egyed, 2012) but we lack large data sets for a systematic comparison. This is also a focus of future work.

9. Future work

We validated our approach using automatically generated tests covering most possible combinations of correct inputs and inputs containing errors. We believe that such a validation is useful to assess scalability, correctness, and completeness. Furthermore we conducted a user experiment which confirm the usefullness of our approach. It would also be interesting to see whether our approach could be a complement to other trace capture techniques such as information retrieval (IR) (Cleland-Huang et al., 2007; Duan and Cleland-Huang, 2007) which are known to contain many erroneous findings. These are future work but it should be noted that the need for a better language is not only motivated by our useability study but also by a series of experiments (Egyed et al., 2010; Mäder and Egyed, 2012) where it was observed that engineers often have difficulties in correctly and completely identifying traces. The present language is the result of several years of observations involved both experienced and inexperienced subjects. The key observation is that a trace matrix is not an ideal medium for capturing traceability. Our language is meant to be the beginning. As two subjecs in our useability study suggested, other constructs could be useful also. For instance two developers requested having the constructs defined in both directions: from requirements to code and from code to requirements, which is not supported at the moment. As part of our future work, we intend to conduct a more elaborate experiment on a longer span of time with more participants. The partner company implementing CMA will continue to use our approach.

10. 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) (Cleland-Huang et al., 2007; Duan and Cleland-Huang, 2007) 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 (Antoniol, 2001; Egyed and Grunbacher, 2002; Murphy et al., 1995), code and documentation (Marcus and Maletic, 2003), architecture and test cases (Muccini et al., 2004), architecture and code (Murta et al., 2008), or features and code (Dagenais et al., 2007)]. Researchers have proposed various techniques and heuristics to support the automation of trace recovery. Examples include event-based approaches (Cleland-Huang et al., 2003), information retrieval (Cleland-Huang et al., 2007; Duan and Cleland-Huang, 2007), feature location techniques (Koschke and Quante, 2005), process-oriented approaches (Pohl, 1996) scenario-based techniques (Egyed, 2003), or rule-based methods (Spanoudakis et al., 2004). 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 (Gotel and Finkelstein, 1994; Lindvall and Sandahl, 1996; Neumuller and Grunbacher, 2006) The approaches of Haumer et al. (1999), Jackson (1991), and Cox et al. (2001) 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 (Koschke and Quante, 2005). Pinheiro and Goguen (1996) 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 (Antoniol, 2001). Spanoudakis et al. (2004) 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 (Cleland-Huang et al., 2007). A forward engineering approach is taken by Richardson and Green (2004) 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) (Cleland-Huang et al., 2005; Eaddy et al., 2008). 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 (Abbattista et al., 1994; Briand et al., 2003; Lee et al., 2000; Tonella, 2003) or understanding the conflict and cooperation among requirements (Egyed and Grunbacher, 2004). 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.

11. Conclusion

This paper presented an extension to our approach to trace discovery and validation. Our approach expects the engineer to define assumptions on artifact-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 shields 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.

Acknowledgement

We gratefully acknowledge funding from the Austrian Science Fund (FWF): P 23115-N23.

References

Drools, http://www.jboss.org/drools/. (Online; accessed 21.01.14) (2014).
Gantt Project, http://www.ganttproject.biz/. (Online; accessed 21.01.14). (2014).
JHotDraw, http://www.jhotdraw.org/. (Online; accessed 21.01.14). (2014).
ReactOS, http://www.reactos.org/. (Online; accessed 21.01.14). (2014).
Abbattista, F., Lanubile, F., Mastelloni, G., Visaggio, G., 1994. An experiment on the effect of design recording on impact analysis. In: Proceedings of the International Conference on Software Maintenance, pp. 253–259. doi:10.1109/ICSM.1994.336769.
Antoniol, G., 2001. Design-code traceability recovery: selecting the basic linkage properties. Sci. Comput. Program. 40 (2-3), 213–234. doi:10.1016/S0167-6423(01)00016-8.
Bianchi, A., Fasolino, A., Visaggio, G., 2000. An exploratory case study of the maintenance effectiveness of traceability models. In: Proceedings 8th International Workshop on Program Comprehension. Limerick, Ireland, pp. 149–158. doi:10.1109/WPC.2000.852489.
Biere, A., 2008. Picosat essentials. In: Journal on Satisfiability, Boolean Modeling and Computation (JSAT), 4. Delft University, pp. 75–97.
Briand, L., Falessi, D., Nejati, S., Sabetzadeh, M., Yue, T., 2014. Traceability and SysML design slices to support safety inspections: a controlled experiment. ACM Trans. Softw. Eng. Methodol. 23 (1), 9:1–9:43. doi:10.1145/2559978.
Briand, L.C., Labiche, Y., O'Sullivan, L., 2003. Impact analysis and change management of UML models. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, p. 256.
Briand, L.C., Labiche, Y., O'Sullivan, L., SÃşwka, M., 2006. Automated impact analysis of UML models. J. Syst. Softw. 79 (3), 339–352. doi:10.1016/j.jss.2005.05.001.
Clarke, S., Harrison, W., Ossher, H., Tarr, P., 1999. Subject-oriented design: towards improved alignment of requirements, design, and code. SIGPLAN Not. 34 (10), 325–339. doi:10.1145/320385.320420.
Cleland-Huang, J., Chang, C., Christensen, M., 2003. Event-Based traceability for managing evolutionary change. IEEE Trans. Softw. Eng. 29 (9), 796–810. doi:10.1109/TSE.2003.1232285.
Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., Christina, S., 2005. Goal-centric traceability for managing non-functional requirements. In: Proceedings. 27th International Conference on Software Engineering, ICSE, pp. 362–371. doi:10.1109/ICSE.2005.1553579.
Cleland-Huang, J., Settimi, R., Romanova, E., Berenbach, B., Clark, S., 2007. Best practices for automated traceability. Computer 40 (6), 27–35. doi:10.1109/MC.2007.195.
Cox, L., Harry, D., Skipper, D., Delugach, H.S., 2001. Dependency analysis using conceptual graphs. In: Proceedings of the 9th International Conference on Conceptual Structures, ICCS 2001. Springer, pp. 117–130.
Dagenais, B., Breu, S., Warr, F., Robillard, M., 2007. Inferring structural patterns for concern traceability in evolving software. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, New York, NY, USA, pp. 254–263. doi:10.1145/1321631.1321669.
Dohyung, K., Java MPEG player. http://peace.snu.ac.kr/dhkim/java/MPEG/.
Duan, C., Cleland-Huang, J., 2007. Clustering support for automated tracing. In: Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering. ACM, New York, NY, USA, pp. 244–253. doi:10.1145/1321631.1321668.
Eaddy, M., Aho, A., Antoniol, G., Guéhéneuc, Y., 2008. CERBERUS: tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: The 16th IEEE International Conference on Program Comprehension. Amsterdam, The Netherlands, pp. 53–62. doi:10.1109/ICPC.2008.39.

Egyed, A., 2003. A Scenario-Driven approach to trace dependency analysis. IEEE Trans. Softw. Eng. 29 (2), 116–132. doi:10.1109/TSE.2003.1178051.
Egyed, A., 2004. Resolving uncertainties during trace analysis. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 3–12. doi:10.1145/1029894.1029899.
Egyed, A., Biffl, S., Heindl, M., Grünbacher, P., 2005. Determining the cost-quality trade-off for automated software traceability. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 360–363. doi:10.1145/1101908.1101970.
Egyed, A., Graf, F., Grünbacher, P., 2010. Effort and quality of recovering requirements-to-code traces: Two exploratory experiments. In: Requirements Engineering Conference (RE), 18th IEEE International. Sydney, NSW, pp. 221–230.
Egyed, A., Grunbacher, P., 2002. Automating requirements traceability: beyond the record & replay paradigm. In: 17th IEEE International Conference on Automated Software Engineering. IEEE, pp. 163–171. doi:10.1109/ASE.2002.1115010.
Egyed, A., Grunbacher, P., 2004. Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. Software, IEEE 21 (6), 50–58. doi:10.1109/MS.2004.40.
Ghabi, A., Egyed, A., 2012. Exploiting traceability uncertainty between architectural models and code. In: Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 171–180. doi:10.1109/WICSA-ECSA.212.25.
Gotel, O., Finkelstein, C., 1994. An analysis of the requirements traceability problem. In: Proceedings of IEEE International Conference on Requirements Engineering. Colorado Springs, COUSA, pp. 94–101. doi:10.1109/ICRE.1994.292398.
Haumer, P., Pohl, K., Weidenhaupt, K., Jarke, M., 1999. Improving reviews by extended traceability. In: Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on, Track3, p. 10. doi:10.1109/HICSS.1999.772891.
Jackson, J., 1991. A keyphrase based traceability scheme. In: IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design, pp. 2/1–2/4.
Koschke, R., Quante, J., 2005. On dynamic feature location. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. Long Beach, CA, USA, p. 86. doi:10.1145/1101908.1101923.
Lee, M., Offutt, A., Alexander, R., 2000. Algorithmic analysis of the impacts of changes to Object-Oriented software. In: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00). IEEE Computer Society, Washington, DC, USA, p. 61.
Li, C.M., Manyà, F., 2009. MaxSAT, Hard and Soft Constraints. In: Handbook of Satisfiability. IOS Press, pp. 613–631.
Lindvall, M., Sandahl, K., 1996. Practical implications of traceability. Softw.: Pract. Exp. 26 (10), 1161–1180. doi:10.1002/(SICI)1097-024X(199610)26:10<1161::AID-SPE58>3.0.CO;2-X.
Mäder, P., Egyed, A., 2011. Do software engineers benefit from source code navigation with traceability? an experiment in software change management. In: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 444–447. doi:10.1109/ASE.2011.6100095.
Mäder, P., Egyed, A., 2012. Assessing the effect of requirements traceability for software maintenance. In: 28th IEEE International Conference on Software Maintenance ICSM, Trento, Italy, pp. 171–180. doi:10.1109/ICSM.2012.6405269.
Marcus, A., Maletic, J.I., 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 125–135.
Muccini, H., Inverardi, P., Bertolino, A., 2004. Using software architecture for code testing. IEEE Trans. Softw. Eng. 30 (3), 160–171. doi:10.1109/TSE.2004.1271170.
Murphy, G.C., Notkin, D., Sullivan, K., 1995. Software reflexion models: bridging the gap between source and high-level models. In: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering. ACM, New York, NY, USA, pp. 18–28. doi:10.1145/222124.222136.
Murta, L.G., Hoek, A., Werner, C.M., 2008. Continuous and automated evolution of architecture-to-implementation traceability links. Automated Software Engg. 15 (1), 75–107. doi:10.1007/s10515-007-0020-6.
Neumuller, C., Grunbacher, P., 2006. Automating software traceability in very small companies: a case study and lessons learned. In: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 145–156. doi:10.1109/ASE.2006.25.
Nöhrer, A., Biere, A., Egyed, A., 2012. Managing SAT inconsistencies with HUMUS. In: VaMoS, pp. 83–91.
Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM 15 (12), 1053–1058. doi:10.1145/361598.361623.
Pinheiro, F.A.C., Goguen, J.A., 1996. An Object-Oriented tool for tracing requirements. IEEE Softw. 13 (2), 52–64. doi:10.1109/52.506462.
Pohl, K., 1996. PRO-ART: enabling requirements pre-traceability. In: Proceedings of the Second International Conference on Requirements Engineering, pp. 76–84. doi:10.1109/ICRE.1996.491432.
Richardson, J., Green, J., 2004. Automating traceability for generated software artifacts. In: Proceedings of the 19th IEEE international conference on Automated software engineering. IEEE Computer Society, Washington, DC, USA, pp. 24–33. doi:10.1109/ASE.2004.20.
Spanoudakis, G., Zisman, A., Perez-Miñana, E., Krause, P., 2004. Rule-based generation of requirements traceability relations. J. Syst. Softw. 72 (2), 105–127. doi:10.1016/S0164-1212(03)00242-5.
Tonella, P., 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. Softw. Eng. 29 (6), 495–509. http://doi.ieeecomputersociety.org/10.1109/TSE.2003.1205178.

Achraf Ghabi M.Sc received his B.S. degree in 2008 and his M.Sc. degree in 2011 in Computer Science both from Johannes Kepler University, Linz, Austria. He is currently a Ph.D. Candidate at the Institute for Software Systems Engineering at the Johannes Kepler University under the mentorship of Prof. Dr. Alexander Egyed. His research interests include requirements engineering, traceability, and change impact analysis.

Prof. Dr. Alexander Egyed heads the Institute for Software Systems Engineering at the Johannes Kepler University, Austria. He is also an Adjunct Assistant Professor at the University of Southern California, USA. Before joining the JKU, Dr. Egyed worked as a Research Scientist for Teknowledge Corporation, USA (2000–2007) and then as a Research Fellow at the University College London, UK (2007–2008). Dr. Egyed received a Doctorate degree in 2000 and a Master of Science degree in 1996, both in Computer Science, from the University of Southern California, USA under the mentorship of Dr. Barry Boehm. His research interests include software design modeling, requirements engineering, consistency checking and resolution, traceability, and change impact analysis.