Automatically Mining Requirements Relationships From Test Cases

Celal Ziftci
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA, USA
cziftci@cs.ucsd.edu

Ingolf Krüger
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA, USA
ikrueger@cs.ucsd.edu

ABSTRACT

Requirements relationships express conceptual dependencies, constraints and associations among the requirements of a software system, such as dependencies and hint-relations. For stakeholders of a system, it is important and beneficial to identify requirements relationships for system design, maintenance and comprehension tasks. In this paper, we build on existing research and use features, realization of functional requirements in software, to automatically retrieve requirements relationships from existing test cases. We evaluate our approach on a chat system, Apache Pool, and Apache Commons CLI. We obtain precision/recall levels as good as or better than currently existing object-tracing and scenario-analysis based approaches when tested on the same case studies. Furthermore, our approach is resistant to scenario selection, and works for all types of systems with a profiler available, unlike existing techniques.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*elicitation methods, methodologies, tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, reengineering*; D.2.9 [**Software Engineering**]: Management—*life cycle, productivity*

General Terms

Measurement, Performance, Design, Economics

Keywords

requirements analysis, requirements dependencies, requirements hint-relations, testing, program understanding, automated analysis, reverse engineering

1. INTRODUCTION

Requirements relationships express conceptual dependencies, constraints and associations among the requirements of a software system [25]. Examples of these relationships are dependencies, a requirement needed by another one; and hint-relations, a requirement being frequently used in conjunction with another requirement [25, 11, 9] (hence, when used, it hints at the other requirement).

Identification of requirements relationships in a system's domain is important in several aspects. During the design and maintenance of a system, understanding requirements in isolation is typically not enough, since requirements interact with or depend on each other, and making a change on one requirement may have impact on other requirements [6, 7, 17, 9, 26]. During the design phase, requirements relationships help ensure that the system will behave correctly by exposing potential conflicts between requirements to designers [12]. During maintenance, they help developers determine if a change on the system will create conflicts that didn't exist before [12, 7, 17]. In all stages of the software lifecycle, they help developers comprehend a requirement along with its dependencies and the other requirements it typically interacts with before a maintenance task is performed [35, 16, 23, 26]. Finally, during all stages of the software lifecycle, they help customers see an overview of requirements, support design and architecture decisions about the evolution of the software system and prioritize development efforts [30, 29, 28, 9].

Requirements relationships are typically modeled during the requirements analysis stage. Similar to other software artifacts, such as source code, test cases and documentation, the system's requirements relationships model evolves during maintenance of and updates to the system. The benefits of the knowledge about requirements relationships, described above, are possible only if the requirements relationships model is accurate and up-to-date. Requirements relationship models can be manually maintained as software evolves. However, it is labor-intensive, time-consuming and error-prone to acquire and maintain such software artifacts. Furthermore, without disciplined developers, the model gets out-of-date over time during maintenance [14, 20, 8]. Therefore, retrieving and maintaining the relationships model via an automated process is important and convenient.

In this paper, we propose to automatically reverse engineer requirements relationships using existing test cases, such as unit tests, integration tests acceptance tests and system tests. Testing is an important and critical part of the software development lifecycle. It increases the quality of the system, and it is typically employed by many, if not all, software development teams. Based on empirical studies, in many systems, the amount of test code produced is comparable to the code produced for the system itself, ranging from 50 percent less to 50 percent more [22, 33]. The amount of test code is even more for some systems, such as critical systems or systems built following the Test Driven Development (TDD) process [15]. Therefore, the effort spent on testing and its cost in the development process is substantial. We propose using this investment to reverse engineer requirements relationships from the test cases automatically.

Recent effective methods that automatically determine requirements relationships use dynamic analysis to track the flow of objects at runtime in object-oriented systems [27, 19]. For each requirement, they exercise the system via a *scenario* (an act triggered on the system to exercise a requirement). They profile the system to track the instantiation of objects and how they are passed around in the system as the scenarios for requirements are exercised. If an object that is created by a requirement is used during the execution of another requirement, the latter requirement is said to have a dependency on the former. Although these approaches can successfully find requirements dependencies, they only work for object-oriented systems. Furthermore, they are highly sensitive to the implementation of the system and how objects flow in different parts of the system.

Another family of recent effective methods that determine requirements relationships uses dynamic analysis, scenarios and profiling as described above [12, 23]. They analyze the software components executed, such as classes and methods, while the scenarios for the requirements are exercised. If a requirement contains all of the components exercised by another requirement, the former requirement is said to have a dependency on the latter one [12]. These approaches are not limited to only object-oriented systems, unlike the previous ones. However, they only find requirements dependencies, not hint-relations. Furthermore, they are very sensitive to the selection of scenarios chosen to exercise the requirements.

In this paper, similar to the recent methods described above, we represent functional requirements of software using "features", observable units of behavior of a system that can be triggered by a user [13]. We use scenarios to trigger features, extract programming components (classes, methods, blocks, statements) that can represent each feature, and observe the extracted components in test cases as they execute and find requirements relationships automatically. Our work makes the following contributions:

- A new method to find dependencies and hint-relations between requirements. Our method performs as good as or better than existing techniques on our case studies.
- A new method to find dependencies and hint-relations between requirements that is broader in applicability. Unlike existing methods, our method is not sensitive to the selection of scenarios used to exercise requirements. Furthermore, unlike existing methods, our method is not limited to object-oriented systems; it works for all systems with a profiler available.
- An automated process and tool support to reverse engineer requirements relationships from a software system's test cases as a by-product of automated software development processes, such as TDD and continuous integration.

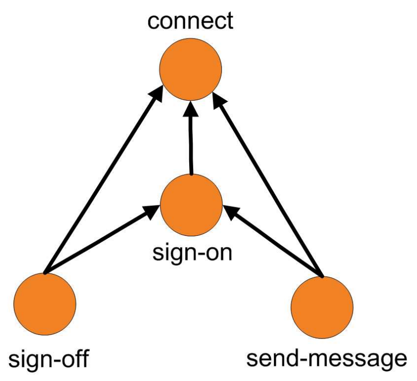


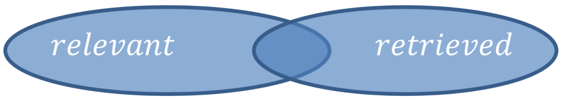
Figure 1: Dependencies between requirements.

2. BACKGROUND

Requirements relationships represent different types of associations between requirements [25]. Two common types of such associations are dependency and hint-relation [25, 11].

Dependency Dependency refers to the case where a requirement must be exercised before another one for the second one to work properly. An example from a Chat System (one of the case studies we use) is the dependency relationship between requirements `connect`, `sign-on`, `send-message` and `sign-off` (shown in Figure 1): `send-message` requires `sign-on`, and `sign-on` requires `connect`, because to sign on and create a session, one needs to first connect to the server; and to send a message to another chat client, one needs to first sign on successfully. Similarly, `sign-off` depends on `sign-on` and `connect`.

Hint-Relation Hint-relation refers to the case where requirements are not necessarily dependent on each other as in the dependency case, but they are typically used together. Therefore, understanding one potentially helps understanding the other, and it becomes easier to estimate the impact of change in one requirement on the other requirements. As an example, consider Apache Pool [2] (one of the case studies we used). This is a library that provides resource pooling, where a limited number of resources exist and the customers of the pool can check-out and check-in objects as they need them. It is typically used to pool resources that are expensive to create and destroy (such as database connections), to avoid the overhead associated with those lifecycle activities. Pool provides a maximum size to limit the number of resources (`max-size` requirement). One can decide on what happens when the pool is full and a consumer asks to check-out a resource, the `action-on-full` requirement, (such as block the consumer call, or throw an exception). Similarly, if "block the consumer's call" is the action chosen, then the pool provides a way to specify for how long to block the call



$$precision = \frac{|relevant \cap retrieved|}{|retrieved|},\ recall = \frac{|relevant \cap retrieved|}{|relevant|}$$

$$f - measure = \frac{2 \times precision \times recall}{precision + recall}$$

Figure 2: Precision is the correctness of the retrieved relationships; Recall is the coverage of the relevant relationships; and F-measure combines precision and recall into a single metric with equal weight.

(`timeout-on-block` requirement). These requirements are not strictly dependent on each other, i.e. setting one does not necessarily require setting the others (since there are defaults for each of them). However, they are closely related to each other, and when a developer tries to understand one of them, knowing that these requirements are related can help him during comprehension, since it is expected that the implementation of these requirements are related to and possibly interleaved with each other.

In this work, we propose to find these two types of relationships automatically from existing test cases. We use precision, recall and the f-measure as the metrics to measure success (see Figure 2). In the context of finding requirements relationships, the "relevant" relationships are the actual ones, i.e. the ones that we try to find. "Retrieved" relationships are what an approach suggests. Precision and recall correspond, respectively, to accuracy and completeness of the retrieved requirements relationships compared to the relevant relationships. Finally, precision and recall are considered equally important and combined into a single metric: f-measure. On finding requirements relationships, it is important to obtain a high f-measure, because that implies finding both a low number of false positives and a low number of false negatives.

3. RELATED WORK

Previous research exists on automatically finding requirements relationships. There are approaches that use executable use-cases, called scenarios, to represent requirements on the actual system [27, 19]. A scenario can be a test case, manual execution of the requirement on the system (such as using a graphical user interface), or any other specification representing a requirement that can be converted into an executable form. As an example, consider the Chat System: `sign-on` is a requirement, and signing on to the server using the graphical user interface of the system is a scenario for that requirement. These approaches execute the scenarios of the requirements on the system. While scenarios are executed, they track object instantiations [27] and object aliasing [19], i.e. how objects are passed in the system between components (classes, methods) using a profiler (or a similar technology). These approaches propose that if an object that is created during the execution of a requirement is used in another requirement during its execution, then there is a direct dependency of the latter requirement on the former. These methods are successful in finding requirements dependencies. However, they have some shortcomings. First, since they track objects, they only work for systems implemented in object-oriented languages; not for other types of languages such as functional or procedural languages. Second, they are sensitive to the implementation of the system since they rely on the flow of objects. They may report non-existing dependencies and hint-relations due to sharing of non-critical utility objects in the system, and they may miss some relationships since not all relationships require sharing or flow of objects in the system.

A different family of approaches to automatically retrieve requirements relationships also uses scenarios to represent requirements. Similar to the previous ones, they execute the scenarios and collect execution-traces as they execute [12, 23]. Unlike the previous methods, they trace the components executed, such as class and method names. They propose that if the components observed in the execution of a requirement are a subset of the components observed in another requirement, the latter requirement is said to have a dependency on the former one. An advantage of these approaches is that they not only work on object-oriented systems, but also other types of systems, because they don't use object instantiation and flow to determine dependency. One of the disadvantages of these approaches is that observing the same components may not be sufficient to conclude the existence of a dependency. If the common methods for the first requirement are executed in a different order than the second one, this may point to a different requirement. Therefore, these approaches can be misguided, since they only analyze the existence of the components, not their order. Another disadvantage of these approaches is that they only detect dependencies; they cannot detect hint-relations between requirements.

A common disadvantage to both of these approaches is in the way they use scenarios. They require executable scenarios for each requirement, which, in some systems, do not readily exist. Furthermore, unless they are supported with manual effort (as in [12]), these approaches can retrieve trace links only for functional requirements of the system. Finally, these approaches might miss some of the trace links, because they only gather trace links on those parts of the system that are exercised by the scenarios. These approaches assume representing each requirement with a single scenario. However, some requirements might be triggered in multiple ways. As an example, consider the Chat System: users are provided a graphical user interface, a command-line client and programmatic access to the server. These approaches will need to choose only one of these as a scenario. Therefore, it is inevitable to miss some trace links for some requirements. This shortcoming can be gapped using an approach as described in [36]. In this approach, a single requirement can be represented with the use of multiple scenarios, which avoids missing some requirements trace links.

Another disadvantage of these approaches is that they are very sensitive to the selection of scenarios. As an example, consider two of the Chat System's requirements: `connect` and `sign-on`. As explained earlier, `sign-on` depends on `connect`. During the selection of scenarios, the scenario for `sign-on` should encapsulate the actions for `connect`, too. Otherwise, neither the objects nor the components executed during its execution will contain the objects or components for `connect`. Therefore, the described approaches will fail to detect the dependency. We propose that this makes these approaches very sensitive to the selection of scenarios.

The method that we propose in this paper builds on existing dynamic analysis based requirements tracing methods to automatically find requirements relationships. Therefore, we discuss the relevant body of research on requirements tracing and dynamic analysis below. We refer the reader to [36] for an in-depth discussion on the comparison of requirements tracing methods, which is out of the scope of this paper.

Requirements tracing is defined as the *"ability to describe and follow the life of a requirement, in both a forward and backward direction"* [5], by *"defining and maintaining relationships to related development artifacts"* [21]. Some examples of these artifacts are source code, test cases and design documents. Our work builds upon the requirements-to-source-code [32, 13, 24] and requirements-to-test-case tracing methods [36]. So these are also described below.

Recent effective requirements-to-source-code tracing methods make use of dynamic analysis and scenarios as described above. They trace the components executed for each requirement while scenarios are running, and they perform different types of analysis to link requirements to source code components (reconnaissance [32], execution slicing [34], formal concept analysis [13], probabilistic ranking [24], footprint graph [12] and many more). The advantage of these approaches is that they are successful in finding the highly relevant source code components for each requirement, which is critical for the work in this paper. The disadvantages of these approaches are the same as for the scenario based ones explained above (since they are all scenario based). In our work, we build upon a well-known method among these approaches [24] (probabilistic ranking) to find source code components that are highly relevant to each requirement. In this approach, components are ranked according to how many times they are observed in the execution trace of each requirement while scenarios are executed, and a probability is calculated for each component (method) that shows how likely it is for a component to represent a requirement (see Section 4.2 for details).

Recent effective requirements-to-test-case tracing methods also use dynamic analysis and scenarios [36]. They first find highly relevant source code components for each requirement. Then they execute the test cases and look for those components in the execution-traces of the test cases. If a component is observed in the execution trace of a test case, they propose that there is a trace link between the requirement and the test case. Since these are also based on the use of scenarios, they also have the disadvantages related to scenarios as described above.

Although not directly related to our work, a significant body of research exists on finding dependencies on the level of software components in source code (classes, methods, blocks, statements). Some of these approaches use static analysis [31, 7, 10], while some use dynamic analysis [35, 16, 17]. Although our work in this paper exhibits some similarities with some of these techniques in the way they perform mathematical analysis on their respective levels (source code), our work is focused on finding requirements relationships, whereas these techniques focus on source code components.

Finally, although not directly related to our work, an approach that finds usage patterns on the application programming interface (API) level is introduced in [18]. In this work, API usage patterns are retrieved from running test cases based on the number of times different API are executed and observed together. This work is again focused on analyzing source code, while our work focuses on requirements.

4. REQRELEX: MINING REQUIREMENTS RELATIONSHIPS FROM TEST CASES

Requirements are what a system needs to provide to its stakeholders. Features, observable units of behavior of a system that can be triggered by a user [13], are the realization of the functional requirements in the system. In this paper, we use the terms *requirement* and *feature* interchangeably.

This work partially builds upon previous work on tracing requirements in test cases [36]. For completeness, we describe the relevant parts of that work (Sections 4.1, 4.2, 4.3) here.

Figure 3 summarizes the inputs, flow and outputs of our approach. We implemented a tool, ReqRelEx, to automate this process. The rest of this section explains the steps of the process in Figure 3 in more detail.

4.1 Features and Scenarios

A **feature** corresponds to the realization of a functional requirement in the system. To invoke a feature, a user needs to trigger it by performing an action. This action can be anything that is executable and exercises the feature on the system. Some examples are: running a test case, using the user-interface of the system, a formal test specification that can be executed. We call such an action that triggers a feature a **scenario** [12, 13]. For a chat system, `send-message` is a feature, and typing a message and clicking "Send" in the user interface is a scenario for `send-message`. Similarly, writing a test case that performs message sending programmatically is another scenario. Since both of these execute the feature, they are both scenarios for `send-message`.

Unless features are already specified in a requirements specification document, finding them is typically a manual task. Similarly, unless requirements specifications are accompanied with directly or indirectly executable scenarios, preparing scenarios is mostly a manual task. An alternative for creating scenarios is to use existing test cases and extract scenarios out of them. These correspond to Step 1 in Figure 3.

4.2 Feature Markers

After features are identified and scenarios are prepared for each feature (Step 1), we execute scenarios and, using a profiler, we collect execution traces for each feature (Step 2 in Figure 3). Execution traces contain information about which components were executed during each scenario. Components can be the combination of file and procedure names for procedural languages; the combination of class names, method names and statement locations for object-oriented

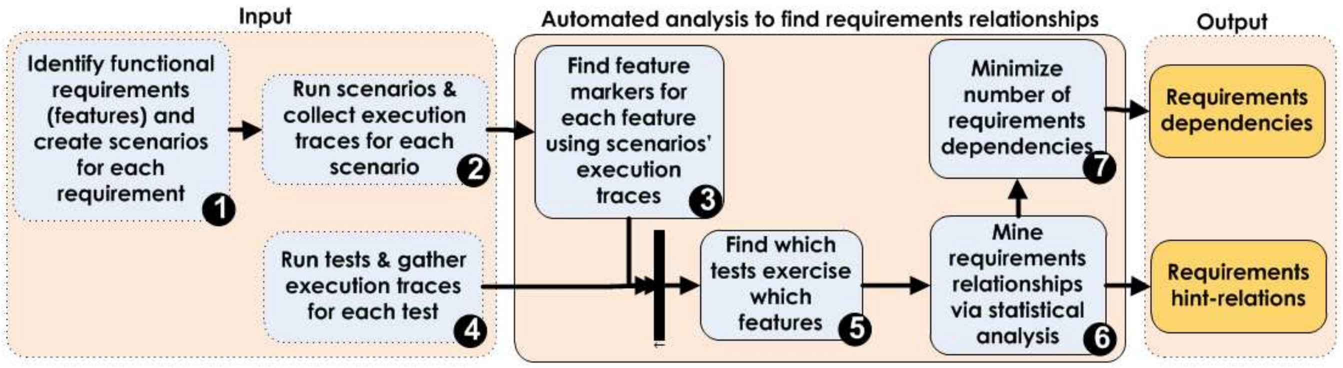

Input
Automated analysis to find requirements relationships
Output
Identify functional requirements (features) and create scenarios for each requirement 1
Run scenarios & collect execution traces for each scenario 2
Find feature markers for each feature using scenarios' execution traces 3
Run tests & gather execution traces for each test 4
Find which tests exercise which features 5
Mine requirements relationships via statistical analysis 6
Minimize number of requirements dependencies 7
Requirements dependencies
Requirements hint-relations

Figure 3: Inputs, outputs and steps of ReqlRelEx. Execution traces of the scenarios and test cases are the inputs, while the requirements dependencies and hint-relations are the outputs.

languages; and combination of namespaces and function names for functional languages (by adjusting the profiler, components can be chosen on different levels of detail according to project properties).

After execution traces for each feature are collected, we find specific components that can represent each feature (Step 3 in Figure 3). To do this, we build upon existing research on finding specific source code locations to understand the implementation of a feature [32, 34, 13, 24]. These techniques aim to find good starting points in source code to comprehend a feature and perform further investigation; and are good at finding components that are specifically related to a feature. Specifically, we use one of these techniques that is known to perform well [24], and find distinguishing components for features which we call **feature markers**.

As an example, below, consider the Chat System, its features and the methods observed in the execution traces of each feature:

$$\begin{aligned}\texttt{connect} &: (m_1, m_2, m_3)\\ \texttt{sign-on} &: (m_3, m_4)\\ \texttt{send-message} &: (m_1, m_3, m_5)\\ \texttt{sign-off} &: (m_3, m_6)\end{aligned}$$

As in [24], we rank each method probabilistically according to the number of features whose execution traces it was observed in. In the above example, m_3 is observed in all four features. Therefore its ranking is $\frac{1}{4}$. Similarly, for all features in our example, the rankings are:

$$\begin{aligned}\texttt{connect} &: (m_1 : 0.5, m_2 : 1.0, m_3 : 0.25)\\ \texttt{sign-on} &: (m_3 : 0.25, m_4 : 1.0)\\ \texttt{send-message} &: (m_1 : 0.5, m_3 : 0.25, m_5 : 1.0)\\ \texttt{sign-off} &: (m_3 : 0.25, m_6 : 1.0)\end{aligned}$$

After this ranking, we use a heuristic strategy (a constant number or a more adaptive strategy for each feature) to choose the highest ranked methods as feature markers for each feature. For our chat system example, if we choose a single feature marker for each feature, they would be:

$$\begin{aligned}\texttt{connect} &: [m_2]\\ \texttt{sign-on} &: [m_4]\\ \texttt{send-message} &: [m_5]\\ \texttt{sign-off} &: [m_6]\end{aligned}$$

This way, we find feature markers to represent all features.

4.3 Finding Which Test Cases Exercise Which Features

After we find feature markers for each feature, we execute the test cases and, again using a profiler, we collect their execution traces (Step 4 in Fig. 2). These execution traces contain the same type of components as for scenarios as described in the previous subsection. Next, we look for the feature markers inside the execution traces of each test (Step 5 in Figure 3). Using the Chat System example, consider the feature marker for `connect` and the execution trace for the test case `testConnect`:

$$\begin{aligned}\texttt{connect} &: [m_2]\\ \texttt{testConnect} &: (m_2, m_4, m_7, m_{13}, m_{146})\end{aligned}$$

Since `testConnect`'s execution-trace contains m_2 (the feature marker of `connect`), we observe that `testConnect` exercises connect. Performing this operation for all features and tests, we find which test cases execute which features (Step 5 in Figure 3).

4.4 Mining Requirements Relationships

Once we find out which test cases execute which features, we investigate the order in which features are observed in each test case. Based on this, we propose the following:

- If a feature f_p is often observed before another feature f_q, we deduce that it is highly likely that f_q depends on f_p.
- If two features f_p and f_q are often executed in the same test case (but in possibly different orders in different test cases), we deduce that it is highly likely that f_p and f_q have a hint-relation, i.e. they do not depend on each other, but they are typically used together.

As an example, consider the feature markers for `connect` and `sign-on`, and the execution trace for the test `testConnectAndSignOn`:

$$\begin{aligned}\texttt{connect} &: [m_2] \\ \texttt{sign-on} &: [m_4] \\ \texttt{testConnectAndSignOn} &: (m_2, m_4, m_8, m_{16}, m_{46})\end{aligned}$$

We observe that in `testConnectAndSignOn`, `connect` is executed before `sign-on` (because m_2 comes before m_4). If this is observed in many other test cases, we propose that `sign-on` likely depends on `connect`. Similarly, if two features are observed in many test cases, but in different orders (i.e. in some, one of them precedes the other, while in others the reverse happens), we propose that they likely have a hint-relation.

In summary, to find such relationships between requirements, we analyze the execution traces of all test cases and look for statistically significant correlations between features observed together in test cases (Step 6 in Figure 3).

4.5 Minimizing the Number of Requirements Dependencies

Once this analysis is performed on the execution traces of test cases, there will likely be many requirements dependencies discovered due to the transitive nature of dependence. As an example, consider the requirements from the Chat System: `connect`, `sign-on`, `send-message`. As described earlier, `send-message` depends on `sign-on`, which depends on `connect`. In such a dependence relationship, it is not necessary to explicitly document that `send-message` depends on `connect`, since that is already implied due to transitivity. In our analysis of the test cases, there will be many such dependencies discovered explicitly (due to the nature of the analysis performed in Section 4.4). Unless such implicit transitive relationships are discarded, there will be an overwhelming amount of information for stakeholders to consume. For this reason, we build a graph on the requirements dependencies we discover, and we apply transitive graph reduction to discard the implicit dependencies (Step 7 in Figure 3). This provides a much clearer picture of the dependencies for developers. Figure 4 shows an example of how the dependencies in Figure 1 look like after reduction.

This reduction is not performed for hint-relations, since hint-relation is not transitive like dependence.

4.6 Tool Support

For our approach, we provide automated tool support that can be integrated into automated software processes (such as continuous build and TDD). As an example, our tool can be run as a task similar to running test cases and providing code coverage in a continuous build system. Execution traces for each feature and execution traces for tests are the inputs to our system. Given these inputs, our tool outputs the requirements relationships. For an example, see Figure 4 where a portion of the automatically retrieved requirements dependencies are output by ReqRelEx for the Chat System.

As described in previous work [36], to make the collection of execution traces of scenarios easier, we propose using automated tests as feature scenarios with a special identifier (*@FeatureScenario("")* in Java). This approach is similar to how automated test cases are typically marked with special identifiers for automated tools (such as *@Test* for JUnit [4] in Java). This way, continuous build systems can run the scenarios and test cases, collect their execution traces, and input them to ReqRelEx automatically.

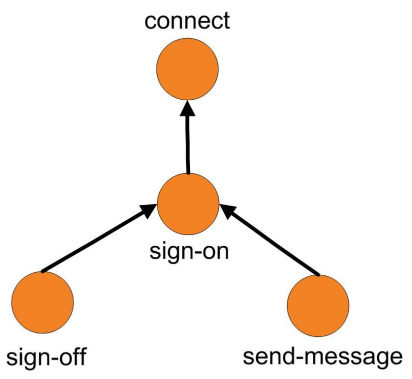


Figure 4: Sample results on minimizing the number of dependencies found in the Chat System via transitive reduction. This graph is a reduced version of the one shown in Figure 1.

Table 1: ReqRelEx case study properties.

Case Study	# Total Relationships	# Dependencies	# Hint relations
Chat System	28	26	2
Apache Pool [2]	9	3	6
Apache Commons CLI [1]	21	3	18

5. EVALUATION

To assess the validity of our approach, we conducted three case studies: a chat system used in teaching a Software Engineering class at UCSD, the open source Apache Pool library [2], and the open source Apache Commons CLI library [1]. All of these systems are implemented in Java, and they have unit and system tests prepared to be run with JUnit [4]. Therefore, they are very suitable for mining requirements relationships using test cases. Table 1 summarizes the statistics relevant to requirements relationships for each project.

To compare our results, we implemented two recent techniques for automatic detection of requirements relationships: object flow analysis of Salah and Mancoridis [27] and Lienhard et al. [19] (which we call Salah's approach in the rest of this paper), and scenario analysis of Egyed [12] (which we call Egyed's approach in the rest of this paper). Salah's approach [27, 19] tracks the flow of objects as scenarios are executed and proposes that: if a feature uses objects created by another feature, there is a dependency between them; if two features share usage of some objects, they have a hint-relation. Egyed's approach [12] analyzes the scenarios themselves: if the scenario of a feature contains all components of another feature, then the former depends on the latter.

We used precision, recall and f-measure as the indicator of success for each method. The rest of this section explains the input preparations of the case studies for ReqRelEx, Salah's approach [27, 19] and Egyed's approach [12].

5.1 Finding Requirements

The Chat System had its requirements documented in a requirements specification document in text form. Apache Pool [2] and Apache Commons CLI [1] did not have requirements specification documents. So we gathered their requirements and their relationships using their webpages online, javadocs, and comments manually. This preparation corresponds to Step 1 in Figure 3, and took about 5 hours for each project. To gather the relationships between requirements, we asked two developers to perform the task and compared their findings to eliminate errors.

It is important to note that, using its documentation or source code, it may not be possible to find all requirements in a software system. Therefore, we can only claim that our analysis of the case studies is partial.

5.2 Creating Scenarios

We prepared scenarios for the chat system manually (Step 1 in Figure 3). We used the existing graphical user interface of the system to perform actions such as `connect` and `sign-on`.

For Apache Pool [2] and Apache Commons CLI [1], we created scenarios as executable test cases using the *@FeatureScenario("")* markers explained in Section 4.6. Each test case was about 2-5 lines of code (considerably shorter compared to the existing tests). For each project, this took about 1 hour.

5.3 Collecting Execution Scenarios

While the scenarios and tests were running, we collected the execution traces using AspectJ [3] (Steps 2 and 4 in Figure 3). Note that AspectJ is not a profiler, but it can be used for this purpose by weaving method entries and printing their names.

We found all test cases that exercise the requirements we identified in Section 5.1 for each case study, and used them in our analysis.

5.4 Inputs to Other Approaches

For Salah's approach [27, 19], we tracked objects when they are instantiated and then used elsewhere using their location in the heap. This ensures that we follow both the flow of objects [27] and any aliasing effects [19].

For Egyed's approach [12], we used the same scenarios and their execution traces that we prepared as input to ReqRelEx.

6. DISCUSSION

Table 2 summarizes the case study results for all approaches: Salah's approach [27, 19], Egyed's approach [12] and ReqRelEx. We provide precision, recall and f-measure values for each approach on the respective types of requirements relationships they support. For ReqRelEx and Salah's approach, we provide results for both requirements dependencies and hint-relations as well as a combination of the two (combined). For Egyed's approach, we only provide requirements dependency results, since it only provides dependency relationships for requirements. During our comparisons, we use the f-measure since it incorporates both precision and recall with equal weight.

Table 2 contains the results of two different experiments we performed: In the first experiment (the first three rows for each case study), the inputs are prepared in conformance to what each technique expects as described in Section IV. In the second experiment (rows four and five for each case study, where each technique is marked with a *), the input is modified to demonstrate a shortcoming of the existing techniques on scenario selection. Each experiment is discussed in detail below.

As discussed in the related work section, Salah's approach [27, 19] might produce false positives since object sharing or flow does not always imply that there are relationships between requirements (but rather in their implementations). It may also result in false negatives, since requirements relationships may exist in systems even in the absence of object interactions. These are observed in our case studies (since precision and recall are not equal to 100%).

Egyed's approach [12] should not have false positives if scenarios are properly selected (hence 100% recall). However, it can produce false positives, since it only investigates the existence of components in execution traces, not the order in which they are executed in scenarios. These are also observed in our case studies.

Below, we provide a discussion of how ReqRelEx compares with Salah's [27, 19] and Egyed's [12] approaches in the first experiment (first three rows for each case study).

First, we compare our results with Salah's approach [27, 19] (rows 1 and 3 in Table 2 for each case study). We provide combined metrics that show how successful each approach is on finding a relationship between requirements without distinction on whether it is a dependency or a hint-relation. Based on the results of the case studies, ReqRelEx performs better overall on finding the relationships before they are categorized as dependency or hint-relation on all case studies.

On finding hint-relations, ReqRelEx performs very close or better on the case studies that have higher number of hint-relations (Apache Pool [2] with 6, and Commons CLI [1] with 18), while it performs worse on the chat system which has only 2 hint-relations. Upon investigation, we observed that ReqRelEx categorizes those two hint-relations as dependencies because the features in both are stylistically used in a fixed order by the developers in test cases, so ReqRelEx misses both of them in this case study. Overall, ReqRelEx achieves comparable or better results compared to Salah's approach [27, 19] since it provides very close or better results on those case studies with a higher number of hint-relations. Overall, however, none of the approaches performs well on

Table 2: Case study results on finding requirements relationships

Case Study	Technique	Results: Combined			Hint-relation			Dependency		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Chat System	1. Salah	40.43	67.86	50.67	5.56	100	10.53	100	57.69	73.17
	2. Egyed	-	-	-	-	-	-	100	100	100
	3. ReqRelEx	78.79	92.86	85.25	0	0	0	100	96.15	98.04
	4. Egyed*	-	-	-	-	-	-	0	0	0
	5. ReqRelEx *	73.68	100	84.85	0	0	0	89.66	100	94.55
Apache Pool [14]	1. Salah	6.1	62.5	11.12	3.66	50	6.82	33.33	100	50
	2. Egyed	-	-	-	-	-	-	66.67	100	80
	3. ReqRelEx	30.43	87.5	45.16	25	83.33	38.46	66.67	100	80
	4. Egyed*	-	-	-	-	-	-	0	0	0
	5. ReqRelEx *	30.43	87.5	45.16	25	83.33	38.46	66.67	100	80
Apache Commons CLI [5]	1. Salah	66.67	19.05	29.63	66.67	22.22	33.33	0	0	0
	2. Egyed	-	-	-	-	-	-	23.08	100	37.5
	3. ReqRelEx	37.14	61.9	46.43	28	38.89	32.56	30	100	46.15
	4. Egyed*	-	-	-	-	-	-	0	0	0
	5. ReqRelEx*	37.14	61.9	46.43	28	38.89	32.56	30	100	46.15

Salah refers to [27, 19], Egyed refers to [11]

* indicates that the same techniques were used with modified scenarios as input for the second experiment

detecting hint-relations (compared to dependencies).

On finding dependencies (first three rows for each case study in Table 2), Salah's approach [27, 19] performs worse on our case studies compared to both Egyed's approach [12] and REQRELEX. Egyed's approach [12] and REQRELEX perform very similarly on all case studies with small differences in their f-measures. Overall, REQRELEX achieves comparable or better results compared to Egyed's approach [12], the state-of-the-art, in our case studies.

Next, we provide the results of another experiment (rows four and five for each case study in Table 2) we performed on the same case studies to show that REQRELEX is resistant to the selection of scenarios, unlike Egyed's approach [12] (we do not provide results for Salah's approach [27, 19] here, because there is no way to perform the experiment without fundamentally changing Salah's algorithm since it depends on object flow). Scenario selection and gathering the execution traces of scenarios are very important steps that determine the success of dynamic analysis techniques. Since scenarios are typically created by developers manually, the process is open to mistakes. Therefore, it would be very beneficial for developers to use a technique that is somewhat resistant to such mistakes. In our experiment, we purposefully modified the collection of the execution traces in our scenarios for each requirement so that dependencies are not explicit right in the execution traces. As an example, consider the two requirements `connect` and `sign-on` from the Chat System: `sign-on` has a dependency to `connect`. Therefore, when the components of the scenarios for these requirements are collected, the execution traces for `sign-on` should contain all components in the execution traces of `connect`. However, when developers create scenarios for requirements and collect execution traces of each scenario, they may only include the parts relevant to `sign-on` in its execution traces (purposefully or by mistake) and leave out the ones relating to `connect`. For Egyed's approach [12] to find the dependencies properly, all execution traces should be collected as described above (i.e. dependent requirements

Table 3: Comparison of the methods compared

	Salah [27, 19]	Egyed [12]	ReqRelEx
Requires scenarios	Yes	Yes	Yes
Works on object-oriented systems	Yes	Yes	Yes
Works on systems that are not object-oriented	No	Yes	Yes
Resistant to scenario selection	—	No	Yes
Works in the absence of test cases	Yes	Yes	No

should explicitly contain the execution traces of their dependencies). For REQRELEX, however, this is not a necessity, because it doesn't analyze the collected execution traces of scenarios to mine requirements dependencies. Instead, it uses those to find feature markers, and uses the test cases to mine the dependencies. Since REQRELEX uses probabilistic ranking on choosing feature markers, we argue that it will likely still choose good components to represent each feature, and be very resistant to such mistakes on scenario selection. Rows 4 and 5 in Table 2 for each case study show the same experiments run again to find requirements relationships. As the case study results suggest, REQRELEX obtains almost the same results, while Egyed's approach [12] fails to find any of the dependencies in this new experiment, as expected.

Finally, Table 3 summarizes the applicability of each approach with pros and cons. Salah's approach [27, 19] only works on object-oriented systems since it relies on object flow and aliasing. On the other hand, Egyed's approach [12] and REQRELEX work for all systems with a profiler available. Egyed's approach [12] is very sensitive to the selection of scenarios, while REQRELEX is resistant to it. And finally, ReqRelEx depends on the existence of test cases, while the other approaches do not.

In the rest of this section, we discuss advantages and limitations of our approach.

Our approach is independent from the programming language with which the system is built, unlike Salah's approach [27, 19] which only works for object-oriented systems. The only requirement of our approach is that it uses a profiler to gather execution traces, and the components in the execution trace are of the same type for tests and scenarios. Our tool currently works for Java, but it is easily extensible to work for any other language (such as object-oriented, functional, procedural) for which a profiler is available (such as C, C++, python, perl, Smalltalk).

An advantage of our approach is that, even though it finds dependencies and hint-relations currently, it can easily be extended to find other types of relationships that can be deduced from test cases. We argue that test cases are a rich source of information that contain implicit knowledge about requirements relationships (such as dependencies and hint-relations as shown in this paper). Therefore, they can be used for mining other types of requirements relationships.

Another big advantage of our approach is that, scenarios can be provided as test cases themselves. When the source code of the system is refactored or changed, developers fix tests to keep them passing after the changes. Since scenarios are also test cases, developers will fix them too and scenarios will always stay up to date. Although this demands some effort from developers, the automatically mined requirements relationships will stay up to date as software evolves.

Another advantage of our approach is that it is resistant to the selection of scenarios. Scenario selection typically determines the success of dynamic analysis techniques, so it is beneficial for developers to use a method that is resistant to the selection of scenarios. This decreases the burden on developers by tolerating some mistakes.

Dependence on test cases might be listed as a disadvantage of our approach. However, although there may be some systems without test cases, we argue that it is commonplace for many production systems to have test cases to ensure correct behavior [22, 33]. Therefore we argue that our technique can still be successfully used for many production systems.

One limitation of our approach is how it uses test cases to find dependencies: If the developers of a test suite have a certain style such that they always exercise a requirement before another, even though there is no dependency between them, ReqRelEx may wrongly deduce that there is a dependency. In fact, this was the reason that ReqRelEx did not obtain 100% precision on the chat system case study. This vulnerability can be fixed by using mutations to change the order of the exercised requirements in the test cases automatically, and checking if dependencies found are actual dependencies. We leave this as future work.

Another limitation of our approach is on finding requirements relationships overall. If the test suite does not contain test cases that exhibit the conceptual relationships (i.e. missing test cases), ReqRelEx will miss them (hence the recall numbers are not 100% in our case studies). Similarly, the success of our approach is limited by the quality and properties of the test cases in the test suite. This is exhibited by the differences in the results across different case studies. These can be partially mitigated by complementing ReqRelEx with one of the existing techniques [12, 27, 19].

Another limitation of our approach is that it only applies to functional requirements currently. Our approach can be complemented with manual effort, as done in [12], to detect relationships for non-functional requirements (robustness, security), which we leave as future work.

Finally, our approach builds upon scenario based dynamic analysis techniques [24]. Therefore, it exhibits the same limitations for these techniques described in the related work section, such as missing coverage. This can be mitigated using multiple scenarios for each requirement [36] to increase the coverage of dynamic analysis.

6.1 Threats to Validity

In this section, we discuss any issues that might have potentially affected our case study results and therefore may limit the interpretations and generalizations of our results.

The first threat is the number and type of the case studies we used and the extent they represent software systems used in practice. The chat system is software used in a class at UCSD, and Apache Pool [2] and Apache Commons CLI [1] are open-source software commonly used in production. We picked our case studies from different domains to mitigate this threat. This threat can be reduced even further if more software systems of varying size from more domains are used for further experiments.

Another threat is the selection of functional requirements and scenarios to obtain execution traces for ReqRelEx. We are not domain experts of the software used in the case studies. Therefore, we cannot claim that we found all requirements and our scenarios capture them best. Thus, depending on the chosen requirements and scenarios, the results may differ.

Another threat is the preparation of the ground truth for requirements relationships in our case studies, which we performed manually. To mitigate risk, two different developers performed these tasks and the results are confirmed comparing their responses. However, it is still possible that mistakes have happened.

7. CONCLUSION

Requirements relationships describe different conceptual dependencies, constraints and associations between the requirements of a system [25].

Determining requirements relationships in a system is important on performing different tasks in the different lifecycle stages of the development of the system. During design and maintenance, requirements relationships help on determining possible requirements conflicts [12], and determining whether making a change on a requirement may have an impact on other requirements [6, 7, 17, 9, 26]. During maintenance, they help developers on program comprehension before a requirement is modified to help them understand the implications and investigate the related requirements that may help during the maintenance activity [35, 16, 23, 9]. During all stages, they help stakeholders see an overview of requirements to help with design and architecture maintenance decisions [30, 29, 28, 9].

The benefits of the identification of requirements relationships, described above, are possible only if the relationships are kept accurate and up-to-date. Acquiring and maintaining the relationships manually is error prone and time consuming [14, 20, 8]. Therefore it would be very beneficial to retrieve and maintain them automatically.

In this paper, we propose retrieving and maintaining two types of requirements relationships automatically using existing test cases: dependencies and hint-relations. A requirement is dependent on another if it requires that requirement to be exercised before itself to behave properly. There is a hint-relation between two requirements if they tend to be used together, but are not necessarily dependent on each other.

Testing is an important part of many software development processes, and a considerable amount of test code is produced in production systems based on empirical studies [22, 33]. The amount of test code is even more for critical systems and systems developed using Test-Driven-Development. Having many test cases increases the investment on testing during the system's development. We propose making use of this investment to automatically mine requirements relationships from existing test cases.

In this paper, we build upon existing literature [12, 13, 24, 36] to trace features in source-code via scenarios. Once the features are traced in source-code, we use highly relevant traces in the source code to find which test cases exercise which features [36]. Finally, after identifying which test cases exercise which features, we perform statistical analysis to find relationships between requirements. We propose that if a requirement is always observed before another one, then there is a dependency of the latter requirement to the former. Similarly, if two requirements are observed in different orders in different test cases, but tend to be observed together many times, we then propose that they have a hint-relation.

Our approach achieves as good as or better precision, recall and f-measure values on the case studies we performed compared to the currently known approaches on finding requirements dependencies and hint-relations [12, 27, 19].

Our approach has many benefits: unlike existing methods [27, 19], it works for both object-oriented systems and for any other type of systems with a profiler available. It is also resistant to the selection of scenarios, unlike existing techniques [12]. It is also fully automated with no need for human intervention during its analysis. The only requirements, similar to existing research [12, 27, 19], are to have a profiler available for the system to be analyzed and the creation of scenarios that represent requirements.

Finally, we propose an automated process and tool support, ReqRelEx, to automatically find requirements relationships as a by-product of automated software development processes such as continuous integration and Test-Driven-Development.

8. FUTURE WORK

If the test-suite was developed stylistically to always exercise two requirements that do not depend on each other in the same order, our technique can be misguided and detect such requirements pairs to have a dependency. This can be mitigated by using controlled mutations to check if an automatically mined dependency is indeed a dependency or not.

Our technique currently only finds two types of relationships. It can be extended to make use of test cases to find more types of requirements relationships.

9. ACKNOWLEDGEMENTS

This work was supported in part by NSF Grants CNS-0932403 and 0729029, as well as a generous donation of phones by Qualcomm.

10. REFERENCES

[1] Apache Commons CLI. http://commons.apache.org/cli. Accessed: 12/01/2013.

[2] Apache Pool. http://commons.apache.org/pool/. Accessed: 12/01/2013.

[3] AspectJ. http://www.eclipse.org/aspectj/. Accessed: 12/01/2013.

[4] JUnit. http://www.junit.org/. Accessed: 12/01/2013.

[5] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, Oct. 2002.

[6] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE, 2005.

[7] H. P. Breivold, I. Crnkovic, R. Land, and S. Larsson. Using dependency model to support software architecture evolution. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 82–91. IEEE, Sept. 2008.

[8] S. Brinkkemper. Requirements engineering research the industry is and is not waiting for. In *Proceedings of the 10th International Workshop on Requirements Engineering: Foundation for Software Quality*, pages 41–54, 2004.

[9] T. B. Callo Arias, P. Spek, and P. Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, 2011.

[10] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Comput. Soc, 2000.

[11] A. G. Dahlstedt and A. Persson. Requirements Interdependencies - Moulding the State of Research

into a Research Agenda. *9th International Workshop on Requirements Engineering Foundation for Software Quality REFSQ03*, pages 55–64, 2003.

[12] A. Egyed and P. Grünbacher. Supporting software understanding with automated requirements traceability. *International Journal of Software Engineering*, 15:783, 2005.

[13] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, Mar. 2003.

[14] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, 1994.

[15] J. H. Hayes, A. Dekhtyar, and D. S. Janzen. Towards traceable test-driven development. In *ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 26–30. IEEE, May 2009.

[16] J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich. Static Execute After/Before as a replacement of traditional software dependencies. In *IEEE International Conference on Software Maintenance*, pages 137–146. IEEE, Sept. 2008.

[17] J. Law and G. Rothermel. Incremental dynamic impact analysis for evolving software systems. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 430–441. IEEE, 2003.

[18] C. Lee, F. Chen, and G. Rosu. Mining parametric specifications. In *Proceedings of the 33rd international conference on Software engineering - ICSE '11*, page 591, New York, New York, USA, 2011. ACM Press.

[19] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking Objects to Detect Feature Dependencies. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 59–68. IEEE, 2007.

[20] M. Lormans, A. van Deursen, E. Nocker, and A. de Zeeuw. Managing evolving requirements in an outsourcing context: an industrial experience report. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 149–158. IEEE, 2004.

[21] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4):13–es, Sept. 2007.

[22] E. M. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Proceedings of the 25th International Conference on Software Engineering*, volume 6 of *ICSE '03*, pages 564–569. IEEE Computer Society Washington, DC, USA, IEEE Computer Society, 2003.

[23] J. L. Pfaltz. Using Concept Lattices to Uncover Causal Dependencies in Software. *4th International Conference Formal Concept Analysis*, 3874:233–247, 2006.

[24] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[25] M. Riebisch. Towards a More Precise Definition of Feature Models. *Modelling Variability for Object Oriented Product Lines*, 22(3):64–76, 2003.

[26] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4):1–36, Aug. 2008.

[27] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *20th IEEE International Conference on Software Maintenance*, pages 72–81. IEEE, 2004.

[28] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. *ACM SIGPLAN Notices*, 40(10):167, Oct. 2005.

[29] J. A. Stafford and A. L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(04):431–451, 2001.

[30] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *Proceedings of the 17th International Conference on Automated Software Engineering*, pages 241–244. IEEE, 2002.

[31] Z. Wei, M. Hong, and Z. Haiyan. A feature-oriented approach to modeling requirements dependencies. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 273–282. IEEE, 2005.

[32] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal Of Software Maintenance Research And Practice*, 7(1):49–62, 1995.

[33] L. Williams, E. M. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering 2003 (ISSRE 2003)*, volume 0 of *ISSRE '03*, pages 34–45. IEEE Computer Society, IEEE Computer Society, 2003.

[34] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology (ASSET'99)*, pages 194–203. IEEE Comput. Soc, 1999.

[35] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*, page 185, New York, New York, USA, July 2007. ACM Press.

[36] C. Ziftci and I. Krueger. Tracing requirements to tests with high precision and recall. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 472–475, Kansas, Nov. 2011. IEEE.