Detecting Implied Scenarios from Execution Traces

Felipe Cantal de Sousa, Nabor C. Mendonça
Mestrado em Informática Aplicada
Universidade de Fortaleza
Fortaleza, CE, Brazil
felipecantal@gmail.com, nabor@unifor.br

Sebastian Uchitel, Jeff Kramer
Department of Computing
Imperial College London
London, U.K.
su2@doc.ic.ac.uk, j.kramer@imperial.ac.uk

Abstract

A scenario describes how one or more system components interact to provide a certain set of functionalities. Because each scenario only represents a partial view of the overall system behavior, scenario-based specifications may hide unexpected interactions (called implied scenarios) which are not part of any scenario individually. Implied scenarios may either indicate gaps in the original scenario specification, or undesired behavior that should be avoided during scenario implementation. This paper presents a reverse engineering environment to support extraction and detection of implied scenarios from execution traces. The main contribution of the paper is to demonstrate how the concept of implied scenarios, thus far restricted to early phases of the software lifecycle, such as requirements elaboration and reliability prediction, can also be applied to support comprehension and testing of existing systems.

1. Introduction

Scenario-based specifications, such as message sequence charts (MSC's) [9] and UML sequence diagrams [13], are increasingly being used as a visual formalism for describing software requirements. A scenario describes how one or more system components, the environment and users work concurrently and interact in order to provide a certain set of system level functionalities. In this way, each scenario provides a partial specification that, when combined with all other scenarios, contributes to the overall system specification. Despite their simplicity and intuitive graphical representation, scenario-based specifications suffer from an intrinsic limitation whose effects have only started to be better understood and investigated more recently. The limitation stems from the fact that a system’s overall behavior is not always entirely reflected in its partial behavior specifications typically expressed using scenarios [24]. Unforeseen combinations of the ways that some components are expected to interact in their participant scenarios may lead to unexpected interaction patterns which are not part of any scenario individually. Such patterns were called implied scenarios in the pioneer work by Alur et al. [1]. That work has subsequently been extended by Uchitel et al. with the notion of positive and negative implied scenarios [21][23], and with a more general approach to specify and detect implied scenarios using an existing model-checking tool [22]. Positive implied scenarios represent valid interaction patterns from the perspective of the system’s intended behavior, which somehow have been overlooked by stakeholders. Negative implied scenarios, on the other hand, represent behaviors that are invalid or undesired, and that must be avoided during implementation. In either case, the ability to reason about the presence of implied scenarios is a fundamental aspect of the requirements elaboration process, be it for helping stakeholders detect and resolve potential gaps in their original scenario-based specifications, be it for increasing their confidence on the specifications at hand [22].

This paper presents a reverse engineering environment to support extraction and detection of implied scenarios from dynamic system information captured during the execution of existing concurrent applications. The environment provides a suite of reverse engineering tools developed or reused specifically with the aim of supporting the different steps necessary to carry out this process. These steps include: (i) instrumented execution of the target application; (ii) scenario extraction and abstraction from the collected execution traces; and (iii) detection and visualization of potential implied scenarios based on the set of extracted scenarios. Even though the possibility of detecting implied scenarios in MSC specifications extracted from execution traces had been
suggested by Alur et al. in their original paper [1], research on implied scenarios has thus far concentrated on design-level specifications aimed at supporting early software development activities, such as requirements analysis [22][10] and system reliability prediction [16]. Therefore, the main contribution of our work is a practical demonstration of how the concept of implied scenarios can also be applied in later development phases, in particular, to support developers in comprehending and testing undocumented yet possible uses of their systems.

The rest of the paper is organized as follows. The next section gives a brief overview of the concept of implied scenarios. Section 3 describes our proposed process and environment for detecting implied scenarios from execution traces. Section 4 presents the results of a case study using an existing e-commerce application. Section 5 then discusses some of the merits and limitations of our approach, which is compared with related work in section 6. Finally, section 7 concludes the paper.

2. Implied scenarios

The presence of implied scenarios (either positive or negative) in a scenario-based specification indicates gaps that are the result of specifying the behavior of a system from a global perspective yet expecting the behavior to be provided in a local component-wise fashion by independent entities with a local view of the system [21]. This means that if the implemented architecture does not provide components with a rich enough local view of what is happening at a system level, components may not be able to enforce the intended system behavior.

Automated support for the detection of implied scenarios is provided by the LTSA-MSC tool [20]. LTSA-MSC is a direct extension of the LTSA model-checker [11], which has been extended with an MSC-like scenario specification language. This language allows one to define high-level MSCs (called hMSCs) from the parallel composition of multiple basic MSCs (called bMSCs). Given a scenario-based specification containing one hMSC and its constituent bMSCs, LTSA automatically checks whether there is any non-specified bMSC that is implied from that specification. When an implied scenario is detected, the tool asks the user to validate the scenario by categorizing it as either positive or negative, or by simply ignoring it. Once the implied scenario is validated, the user can repeat the detection process until no more implied scenarios are found.

Figure 1 shows an example (taken from [22]) of a scenario-based specification for a Boiler Control system where a Control unit operates a Sensor and an Actuator to control the pressure of a steam boiler. A Database is used as a repository to buffer Sensor information while the Control unit queries the Database and performs calculations before sending commands to the Actuator. The specification includes four bMSCs (Initialize, Register, Analysis and Terminate) and one hMSC describing how the four bMSCs are organized in terms of their potential causal relations. Note that, from the perspective of each individual bMSC, all components in that specification
show valid behaviors. However, when that specification is provided as input to the LTSA tool it detects the implied scenario shown on the bottom right of Figure 1. The interaction pattern represented in the implied scenario reveals that the Control unit may receive (potentially out-of-date) pressure information from a previous activation of the Sensor, since it is allowed to query the Database before the Sensor has registered any new pressure information on it. This behavior, which is clearly not specified in any of the four bMSCs, may happen despite the fact that the hMSC specification states that, after initializing Sensor, there must be some data registered into the Database before any queries can be done. A more detailed discussion of the reasons underlying this and other implied scenarios present in this example can be found in [22].

This example shows that even in a toy system such as the one we have discussed in this section, implied scenarios cannot be detected trivially by observing the given scenario specifications. Thus, there is great value in automatically detecting and validating implied scenarios as they force developers to address subtle yet crucial points regarding the concurrent nature of their systems.

3. The proposed process

3.1. Stages

Our proposed process for detecting implied scenarios from execution traces is illustrated in the form of a UML activity diagram in Figure 2. It is decomposed into four main stages, each one carried out using a different tool. They are: (i) selection of scenarios of interest and generation of their corresponding execution traces (lanes 1 and 2 in Figure 2); (ii) extraction of scenario elements from the generated traces (lane 3); (iii) detection of implied scenarios in a scenario-based specification defined over the extracted scenarios (lane 4); and (iv) scenario visualization in the form of UML sequence diagrams (lane 5). This fourth stage is optional, and is aimed at facilitating comprehension of the extracted scenarios using standard UML case tools [14]. The following subsections describe the first three stages of the process in more details.

3.1.1. Scenario selection and generation of execution traces. The process starts with the choice of which scenarios (or sets of system level functionalities) will be executed using an instrumented version of the target application. This choice is usually made with the help of a user or developer of the target application, or of someone with a larger expertise on its domain. Afterwards, the target application is instrumented, so that its execution can be monitored by a tool called Event Monitor. This tool is responsible for monitoring every event or action (object creation, method invocation, etc.) generated or executed by the application. Events classified as part of one or more of
the selected scenarios will be logged by the tool into the scenarios’ corresponding execution traces.

The types of elements to be included in the generated traces are defined based on an execution trace meta-model (see Figure 3) similar to the trace meta-model proposed by Briand et al. for the recovery of UML sequence diagrams [3]. The main element of the meta-model is the class Scenario, which uniquely represents an application scenario monitored by the Event Monitor. Each scenario is associated with a set of instances (class Instance) and a set of messages (class Message) that represent the components participating in the scenario and their invocation relationships, respectively. The Event Monitor logs scenario-specific events using in-memory data structures that can be easily converted and exported to XML documents compatible with our execution trace meta-model.

The main advantage of using a trace meta-model to store trace elements is that it makes it easier to manipulate those elements, independently of the technology used to implement the Event Monitor and of the target application’s programming language.

3.1.2. Scenario extraction. Abstracting execution trace elements into scenarios is a non-trivial process that may vary according to both the characteristics of the target application and of the specific scenario being extracted. Thus, to be successful in this process a developer must have a thorough understanding of the scenario under analysis, including characteristics such as business rules, execution context and the external architectural environment.

Our scenario extraction process consists in interactively applying filters and modifiers to the (potentially large) set of events originally captured in the execution traces generated for each scenario. To this end, we have implemented a scenario extraction tool that allows a developer to interactively modify the message flow of each scenario, until a satisfactory abstraction level has been achieved. Each output produced by the tool corresponds to a new (more abstract) version of the input scenario, which, in the case that the desired abstraction level has not been achieved, can be fed back as input to the tool, so that a new set of filters and modifiers can be applied to it. Using such an incremental refinement approach has the advantage that it allows the developer herself to define the abstraction level that is most appropriate for representing the scenario.

Currently, our scenario extraction tool supports five filter types. These five types, when applied to the elements of given execution trace, produce scenarios represented at five different levels of abstraction. Table 1 describes which trace elements are filtered out in each of these five abstraction levels.

The first abstraction level is intended to better isolate the target application under analysis. It sets the first boundaries for the scenario and guarantees that only those elements most relevant to the application will be extracted. This is done by filtering out all those elements belonging to external packages (i.e., packages that were not exclusively developed for the target application), such as platform packages for IO, network communication, etc.

The second abstraction level is intended to better isolate the scenario of interest from the rest of the application, preserving only those elements that are directly involved in its implementation. At this level we also remove all elements belonging to utility classes, i.e. classes that offer common services to several other classes of the target application and which, consequently, little contribute to the comprehension of the scenario’s specific goals. Our utility removal mechanisms is based on a fan-in/fan-out analysis of the invocation relations between classes elements included in the execution traces, following a strategy similar to the approach proposed by Hamou-Lhadj et al. [5].

The third and fourth abstraction levels filter out further trace elements considered irrelevant from the perspective of component interactions, namely, invocation of class constructors and invocation of inner methods, respectively.
Finally, the fifth abstraction level removes redundant message exchange patterns that may occur inside a scenario. The occurrence of such patterns is usually associated with the use of loop constructs by a component; therefore, their multiple representations are of little value to help understand the scenario’s overall goal.

It is important to emphasize that our scenario extraction process is not rigid, and both the number of abstraction levels as well as the filter types defined for each level can be changed according to the developer’s needs and the target application’s characteristics.

3.1.3. Implied scenario detection. In this stage, the extracted scenarios are exported to an XML-based representation compatible with the MSC input language required by LTSA [20]. Specifically, each scenario extracted in the previous stage is converted to an equivalent bMSC. In addition, a new hMSC is defined describing the potential causal relationships between two or more of the extracted scenarios (bMSCs). As we have mentioned previously, this set of multiple bMSCs and one hMSC define over them constitutes the basis specification upon which the LTSA tool can search for the presence of implied scenarios.

The implied scenario detection process is also carried out incrementally, with the developer interactively modifying or refining the hMSC specification according to the results produced by LTSA. It is important to note that there may be situations in which no implied scenario is detected by the tool. In such cases, the developer may try to modify the hMSC by removing some bMSCs and/or including new bMSCs. Ultimately, the developer may decide to go back to the first stage and extract new scenarios from the application, or to stop the process with no implied scenarios detected. Therefore, choosing a proper hMSC is a key aspect of our approach, since it cannot be fully automated.

3.2. Support environment

Since the LTSA and UML case tools were entirely reused off-the-shelf, we focused our implementation efforts on the Event Monitor and the Scenario Extractor. The Event Monitor was developed using the Java Platform Debugger Architecture (JPDA) [19], which provides a generic distributed framework for implementing customized Java debuggers. In this way, no source or byte code modifications are necessary to instrument the target application; the only restriction being that it must be executed under the control of the JPDA platform. The scenario extractor in turn uses a series of XSLT scripts [25] to filter and convert the extracted scenario elements into the input formats required by the LTSA and UML case tools.

4. Case study

In an initial validation effort, we have used our proposed process and tools to search for implied scenarios in a scenario-based specification extracted from an existing e-commerce application. E-commerce applications are characterized by a high degree of concurrency, especially in their server-side operations that access shared business components. This makes the scenarios extracted from e-commerce applications likely candidates for implied scenarios detection.

The target application used in our experiments was MyPetStore, which is a simplified version, developed by the first author, of the Java Pet Store application developed by SUN [18]. As the name implies, Java Pet Store is a web application for selling animal pets in a fictional electronic pet store. It was developed by SUN as an open source application to illustrate the best practices in J2EE development. In the case of MyPetStore, we have implemented only basic functionalities related to selling of products and creation and maintenance of the clients and products databases. Table 2 presents the complete list of the functionalities and operations implemented in the current version of MyPetStore.

We have chosen to use MyPetStore at the target application for the case study, instead of the original Java Pet Store, due to strictly pragmatic reasons. This is because the former would allow us total control over the scenario extraction process. As we have discussed in section 3, having a deep understanding of how a given scenario is likely to be implemented in the target application’s source code is fundamental for effectively extracting this scenario from the application’s execution trace. Since our main interest was in validating our approach, rather than precisely

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticate user</td>
<td>Login, Logout</td>
</tr>
<tr>
<td>Maintain users</td>
<td>Find, Insert, Update, Delete</td>
</tr>
<tr>
<td>Maintain products</td>
<td>Find, Insert, Update, Delete</td>
</tr>
<tr>
<td>Shopping</td>
<td>ClearShopCart, ListShopCart, CloseShopCart, InsertItemInShopCart</td>
</tr>
</tbody>
</table>
re-documenting an existing application, using MyPetStore as the case study’s target application has proved a cost-effective solution before we try our tools on “real” concurrent applications. Nonetheless, the fact that MyPetStore itself was also developed following all the J2EE best practices recommended by SUN builds our confidence that the results reported in this paper may generalize to other concurrent applications with similar characteristics.

4.2. Scenario selection and generation of execution traces

Due to the concurrent nature of the target application, some of its operations can be executed in the context of different scenarios. For example, the operation CloseShopCart triggers the execution of four scenarios: prepare shopping, authenticate user, verify product and do shopping. These multiple associations amongst the scenarios illustrate the challenges of defining precisely where (or when) each scenario starts and where it ends. They also reinforce the need of having an expert on the target application around to guide the scenario extraction process.

Table 3 describes the six scenarios selected for the case study as well as the set of MyPetStore operations that had to be monitored in order to generate the scenarios’ execution traces.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Monitored operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>doLogin</td>
</tr>
<tr>
<td>logout</td>
<td>doLogout</td>
</tr>
<tr>
<td>prepare shopping</td>
<td>CloseShopCart</td>
</tr>
<tr>
<td>authenticate user</td>
<td></td>
</tr>
<tr>
<td>do shopping</td>
<td></td>
</tr>
<tr>
<td>delete user</td>
<td>Delete (User)</td>
</tr>
</tbody>
</table>

4.2. Scenario extraction

The execution traces generated for each selected scenario were incrementally processed by the scenario extraction tool, using the five filter types defined for each of the five abstraction levels described in the previous section.

At the first level, the tool removed all elements external to the application’s main business packages. At the second level, all elements belonging to application-specific utility packages and packages not directly involved in the implementation of the five scenarios of interest were eliminated. This level managed to precisely isolate only those trace elements that contribute to the scenarios implementation, resulting in a significant reduction (above 85% for all scenarios) in the set of extracted elements compared to the original set present in the trace. For some scenarios that make extensive use of utility services, as it is the case of delete user, this reduction was even higher (above 97% compared to the original trace). The sets of extracted elements for each scenario were further reduced after processing at the other three abstraction levels. However, we do not elaborate on these results here.

Figure 4 shows the final results for the six scenarios extracted from MyPetStore, represented as UML sequence diagrams. Note that each of the six scenarios shows a valid behavior with respect to its local view of the system. Note, also, how the six scenarios share multiple business components, such as UserControllerImpl, which is shared by login, logout and authenticate user, and ShopControllerImpl, which is shared by prepare shopping, authenticate user and do shopping. This was expected, since resource-sharing is characteristic of concurrent applications.

4.3. Implied scenario detection

The six scenarios shown in Figure 4 were first converted to their equivalent bMSC representations, as required by the LTSA tool. Different subsets of these six bMSCs were then used for defining a number of distinct hMSCs, each one establishing a slightly different organization over those scenarios in terms of the potential runtime relationships. In the end, we were successful in detecting the presence of at least three implied scenarios from MyPetStore. Due to space restrictions, here we will describe only one of those three implied scenarios, which we call “unexpected logout”.

Figure 5 shows the hMSC and the UML sequence diagram for that implied scenario. The hMSC is defined over five of the six scenarios extracted from MyPetStore, namely login, logout, prepare shopping, authenticate user and do shopping. It includes a default initialization scenario, called init, whose single transition leads to the login scenario. From login there is also a single transition, leading to prepare shopping.

1 We were unable to run LTSA with hMSCs containing all the six scenarios due to scalability issues (in many occasions the tool did not respond in a reasonable amount of time, probably due to state explosion in the implied scenario detection process, and had to be interrupted).
From that scenario the hMSC defines three possible transitions: one for the logout scenario, which would force the system to go back to the init scenario; one for the authenticate user scenario, meaning that system would carry on with the current shopping; and one back to prepare shopping, indicating that the user would be allowed to start a new shopping to replace the current one. Once the system reaches the authenticate user scenario, there is only one possible transition, i.e. to do shopping, and from there back to prepare shopping, starting a new shopping cycle. This cycle (prepare shopping → authenticate user → do shopping → prepare shopping) is part of the CloseShopCart operation, in which the three scenarios must always be executed in that strict order.

The “unexpected logout” implied scenario detected in that hMSC (see the right side of Figure 5) has the following execution context:

i. The user successfully logs in to the system (messages 1 and 1.1);
ii. The user prepares a new shopping (message 2);
iii. The system successfully authenticates the user (messages 3 and 3.1);
iv. The user leaves the system (message 4).

Clearly, this sequence of interactions is not specified in any of the five original scenarios individually. In addition, that sequence is not evident from the hMSC specification either, as the hMSC states that the authenticate user scenario (messages 3 and 3.1 in the implied scenario) should be followed by the do shopping scenario, instead of the logout scenario.

Depending on how the system implements the doLogout operation, this implied scenario could be classified as negative, since it may lead to potential erroneous situations from the perspective of the application’s overall behavior. Specifically, where one would normally expect the execution of the do shopping scenario, the implied scenario reveals that there is also the possibility (due to the concurrent nature of application components) of the system executing the logout scenario. This means that, in the case that the implied scenario is actually executed, the application could be forced to interrupt an ongoing shopping transaction abruptly, possibly putting at risk the integrity of its business data. Another problem, of even greater impact, would occur if the application proceeded with the do shopping scenario even after the doLogout operation had been executed. This problem could happen, for instance, in some concurrent architectures where all components are executed using independent threads. In this case, the application would allow the purchase of a shopping item even after the user that started the shopping had left the system, thus indicating a security risk.

The problems discussed above are representative of the benefits that the ability to detect implied scenarios
can bring to later software development activities. For example, to avoid the consistency and security issues raised by the detection of the “unexpected logout” implied scenario, MyPetStore developers could implement a simple test in the beginning of the `doLogout` operation. That test would check whether there is any ongoing shopping transaction that has been started by the current user. In the case that such a transaction exists, the `doLogout` operation could mask the unexpected logout problem by postponing the logout action until all transactions associated with the current user have been completed. Alternatively, the system could ask the user to confirm the logout operation, in which case the user would be warned that all her ongoing transactions would have to be cancelled.

5. Discussion

In addition to contributing to improve the quality of the target system’s documentation, detecting implied scenarios in scenario-based specifications generated from execution traces can also help developers to plan for and, if necessary, to address potential concurrency problems in advance, thus also improving the overall quality of the development process.

Even in the case that a concurrent application under analysis already implements solutions to handle all the potential problems that could be associated with the presence of a particular implied scenario, explicitly detecting that implied scenario from the application’s execution traces would still be useful as a way to draw attention of the developers to critical aspects of the application which otherwise could go unnoticed. This can be even more important during maintenance activities involving new developers that may be unfamiliar with the application at hand.

Knowledge on the presence of implied scenarios can also be useful to support test activities. For example, given a set of execution traces that have been generated from a set of test cases, an implied scenario would be an indication of system behavior that is possible but that was either not covered by the test cases or not produced by running them (because an alternative path occurred due to non-deterministic behavior of the concurrent system). In essence, the implied scenario would give an indication of unexplored system behavior. This may be particularly helpful in test-driven development approaches [2], where implied scenarios could be seen as indications as to whether more testing is needed in a particular direction.

On the other hand, it is also important to emphasize that the size and complexity of the extracted scenarios (in terms of the number of participant components and their message exchanges) can be a limiting factor to our approach. This is because LTSA was originally developed to analyze requirements-level scenario specifications, where a scenario is usually restricted to a few components and message elements. As we have seen during the case study, the tool does not scale well to implementation-level scenarios specifications, where a scenario may contain up to tens or even hundreds of elements. In our work, we tackle the scenario explosion problem in two fronts: (i) by defining event filters to reduce the size of the execution traces generated by the Event Monitor; and (ii) by defining execution trace filters and modifiers to be applied interactively during the scenario extraction process (in the case study, those filters resulted in an average reduction of 90% in the size of the extracted scenarios).

Finally, difficulties in abstracting way execution trace elements reinforce the need to have an application expert always around to guide the scenario extraction process. As we mentioned previously, a more detailed knowledge about the application under...
analysis is crucial to help identify the most relevant trace elements for each selected scenario, and also to guarantee that the extracted scenarios indeed reflect the target application’s implemented functionalities.

6. Related work

Many commercial reverse engineering tools (e.g. Jinsight [8]) use dynamic analysis techniques to recover some form of execution scenarios, usually represented as UML sequence diagrams or MSCs, from a running application. This approach has also been used in a number of other research works (e.g. [3][7][15]). More specifically, Briand et al. [3] propose a tool and an associated methodology for the reverse engineering of UML sequence diagrams. That work uses aspect-oriented programming as the mechanism to instrument the target application, and defines trace execution metamodels to map trace execution elements to their corresponding sequence diagram elements. Hamou-Lhadj et al. [7] describe an environment for recovering behavior models from execution traces. A distinguished feature of that work is the use of fan-in analysis for the removal of elements belonging to utility classes. Richner and Ducasse [15] present a tool for interactive recovery of dynamic information about generic collaborations and roles, which can be applied to any object-oriented programming language. Other researchers have used static analysis techniques to recover conservative sequence diagrams directly from source code artifacts. This is the case of Rountev et al. [17], which propose a technique to map control-flow graphs to the new control structures that have been introduced in the UML2.0 specification; and Mansurov and Campara [12], which present a way to statically extract scenarios represented as MSCs. Finally, motivated by the need to facilitate comprehension and visualization of execution traces generated via dynamic analysis, some researchers have proposed filtering techniques specifically aimed at reducing their size and complexity (e.g. [5][6]).

All the reverse engineering approaches described above focus exclusively on the technology necessary to extract sequence diagrams (or other similar representations) from execution traces or source code artifacts. To the best of our knowledge, no reverse engineering tool has yet been proposed to detect implied scenarios in scenario-based specifications extracted from execution traces. On the other hand, existing work on automatic detection of implied scenarios is restricted to analyzing requirements-level specifications (e.g. [20]), which are of a completely different magnitude (in terms of both size and complexity) than scenarios extracted from execution traces. As we have mentioned along the paper, our work draws upon results from these both research areas. In this regard, the main contribution of this paper is to show how modern scenario recovery techniques can be integrated with an existing implied scenario detection tool, so that the concept of implied scenarios originally proposed by Alur et al. [1] can also be used to support later software development activities.

7. Conclusions and future work

The ability to detect and reasoning about the presence of implied scenarios in an existing concurrent application can offer software developers with a number of interesting benefits. In particular, it promotes application (re)documentation; it gives insights on how the application’s intended behavior is realized in terms of its implemented operations; it forces developers to consider crucial aspects of the application that otherwise could have been overlooked; and it supports application comprehension and testing by anticipating potential concurrency problems, thus allowing those problems to be more effectively identified, tracked and addressed.

As regards future work, we plan to investigate the extent to which the results reported in this paper may generalize to other concurrent applications of varied sizes and domains. To this end, we are currently working on improving the usability and performance of our reverse engineering tools, so that they can be applied to moderate-size concurrent systems without demanding a significant effort from the user. Of particular interest is the design of novel filtering techniques to further reduce the complexity of execution traces and, consequently, to control scenario explosion during the recovery process. This research line is vital to make our tools scale to “real” applications. We are also interested in gaining more insights into the types of knowledge or experience about the subject application that are needed by software engineers to effectively benefit from our approach.

8. References


