Constructing Usage Scenarios for API Redocumentation

Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systä
Tampere University of Technology, Institute of Software Systems
P.O. Box 553, FIN-33101 Tampere, Finland
{juanjuan.jiang, johannes.koskinen, anna.ruokonen, tarja.systa}@tut.fi

Abstract

Software development relies heavily on reusable libraries and software components. For correct use of the provided API, proper documentation is needed. API usage is often demonstrated by giving example applications and code samples. In this paper we propose an approach for mining such usage scenarios from run-time communication between sample applications and the API. This is done automatically by first monitoring the API usage of sample applications, then filtering the generated traces, and finally synthesizing the sequence diagrams and illustrating them in a well-formed way as UML2 sequence diagrams. Such usage scenarios support the software engineer in comprehending the usage of the API. With proper tool support they can also be used for validating other applications' API usage and for generating code for a new application using the same API.

1 Introduction

Today's applications are often constructed relying on reuse, e.g in terms COTS component integration, library reuse, or use of application frameworks. Reusing such existing solutions requires understanding the API or framework interface offered. For correct reuse, the software engineer typically needs to understand both structural and behavioral aspects of the API. From a structural side, she needs to understand what services are available. From a behavioral side, she needs to understand in which order certain services should be called to ensure the desired functionality.

Framework and API documentation typically lists the services available for the reuser. Such documentation describes *what* methods can be called, for example, what are the abstract classes for which implementations needs to be given. To answer *how* a certain functionality can be implemented using the API, code examples are sometimes given; they describe what methods and in which order are to be called for implementing a certain functionality. The code examples thus imply API *usage scenarios* that describe either a recommended or required sequence of API calls needed for implementing this functionality. For example, different usage scenarios can be given to capture rules for implementing different Web service client applications when using a selected Web service toolkit API.

While code fragments can be very useful, they tend to encourage the software engineer to follow a copy-and-paste type of coding practise, without emphasizing the relevant interactions between the application to be implemented and the API or how to separate the required orders of invocations from the freely choosable ones. Koskinen *et al.* propose in [11] an approach and tool support for generating code, in a user-assisted way, from usage scenarios depicted as UML2 sequence diagrams. Such graphical usage scenarios also support the comprehension of the API and they can be used to distinguish alternative ways to use an API or define required orders of the significant API calls.

In the approach discussed in [11], the usage scenarios are assumed, the focus being on their utilization. API documentation, however, does not typically include such usage scenarios and their manual construction requires good knowledge of the API. In this paper, we propose an approach for automated redocumentation of the API by comparing the run-time behavior of sample applications, requiring no information of the API beforehand. The resulting usage scenarios visualize the sample applications' common and different ways of using the API. The approach also supports construction of new or updating existing documentation. Finally, the usage scenarios can be used later on for validation purposes: constructed usage scenarios can be used for validating other applications [9].

2 Related Work

Various tools and methods have been proposed to support run-time analysis of software systems (e.g. [8, 13, 14, 16]). These tools often extract event traces, which include method invocations and possibly also object creations and destructions. They use different kinds of scenario notations for illustration. One of the main challenges in these approaches is to find illustrative visualization methods that both scale up for the huge amount of information extracted and support the user to focus on essential pieces of information she is interested in.

One approach to manage the trace explosion problem is to search "behavioral patterns", i.e. exactly or almost similar sequences of events, from the trace and use them to abstract and/or filter the scenarios [2, 8, 15, 16]. These methods provide only partial support for the problem, but they at least support identification and comprehension of relevant behavioral patterns. UML2 sequence diagram notation, which is also used in our approach, allows illustration of such patterns with its loop and referenced subscenario structures.

By structuring the event trace with behavioral patterns, only limited support for the problem of event trace explosion can be provided. Therefore, we also use another technique, namely filtering. It supports understanding and visualization of the essential run-time behavior by reducing traces to include only the essential behavioral information. This can be done e.g. by instructing the tracer to focus on certain parts of the software only. This useful feature is implemented in the most currently available tracers, e.g. in TPTP [4] that is also used in this work. To guide the filtering, we use a specific filtering model. In a general case, the challenge in this approach is to identify the parts of software that are essential to be included in the trace. For certain specific purposes, however, such parts can be easily identified. This is also the case when monitoring the API usage; communication between an application and an external API can be distinguished from the interactions within the framework and within the application.

The relevant parts of the event trace can also be identified based on predefined rules. Such an approach and tool support has been introduced and demonstrated in [9]; so-called "behavioral profiles" are used to capture essential behavioral rules on the subject software and to guide the filtering of the extracted trace to only contain information relevant from the point of view of these rules. In a study presented in [9], the power of this mechanism was demonstrated; about 99% of the original trace, not relevant from the point of view of the behavioral rules, was filtered out. The behavioral profiles in [9] are given as stereotyped UML2 sequence diagrams. They are thus notationally close to the usage scenarios discussed in this paper. Integration of these two approaches is part of our future work.

Reverse engineering UML2 sequence diagrams from execution traces has been studied also e.g. by Delamare *et al.* in [3], where the construction of combined fragments (loops and alternative fragments) is guided by annotating simple, flat sequence diagrams with state information. In our approach, we conclude such fragments from a state machine, synthesized either automatically or in a user-assisted way. In another approach proposed by Gueheneuc and Ziadi [6], combined fragments are used to combine the basic, flat sequence diagrams, without examining the similarities and difference of the internals (message sequences) of the basic sequence diagrams. We, instead, merge the basic sequences and conclude their common, alternative, and optional parts.

3 Approach

To analyze the usage of an API[1] without having actual application code, a black-box method and dynamic analysis can be used; the information traced when running the application can be used instead of the source code. The programs may (and should) use the API differently, thus allowing the analyzer to collect different usages of the API. Naturally, to get correct usage information, the running programs should use the API in a valid way.

We next introduce an approach to analyze the use of an API using trace information, which is gathered by monitoring the run-time behaviors of a set of applications using the API. The approach is based on UML notation.

3.1 Overview of the Approach

Fig. 1 presents the process flow for the approach. We use a state machine synthesis approach to merge the information presented in execution traces. Common, optional, and alternative parts in these traces are then concluded from the state machine and visualized using the control structures, called *combined fragments*, of UML2 sequence diagram notation. The process can be divided into following steps:

1. **Tracing** applications that use the subject API.
2. **Filtering** the traces and categorizing participants (i.e. message senders and receivers) to *Application* and *API* groups.
3. **Merging** the filtered traces into a state machine.
4. **Transforming** the state machine into usage scenarios. This is done with the aid of information used in **Filtering** step.

In the tracing step, the execution traces are generated by a tracing tool or using instrumented code. Of course, the tracing tool can vary depending on the system, programming language and platform used. Because trace files are usually huge with thousands of messages, the traces are used directly without first loading them to the CASE-tool. In the following subsections, we will clarify the rest of the steps in more detail.

[1]While we use term *API* through the paper, the approach can also be used to analyze e.g. protocols and frameworks.

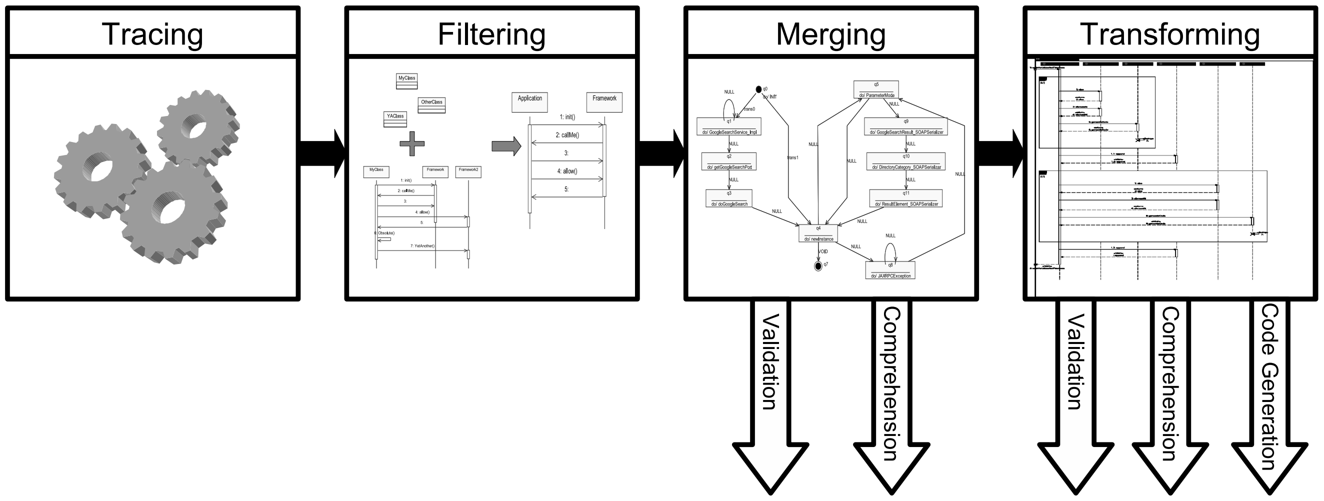

Figure 1. Process Overview

3.2 Filtering Trace Information

Identifying the relevant information and distinguishing that from the typically large amount of uninteresting information included in the traces are challenging tasks. Visualization techniques, typically used in the dynamic analysis approaches and tools [8, 13, 14, 16], provide only limited support for overcoming this problem. Often, e.g. when gathering trace information for our purposes, capturing only a specific part of the run-time behavior is needed.

In addition to limit the amount of trace information, we have to somehow distinguish the application and API parts. To do that, we use a UML model (called *filtering model*) to specify interesting classes and operations, and the roles they play. Further, the classes are divided into two packages (marked with stereotypes «API» and «APP»). Such model can quite easily be created using reverse engineering tools. Naturally, the filtering and categorization can also be done by using textual lists or naming conventions. Still, the model will also give more information about the structural, static side of the analysed API.

Using the filtering model we then automatically filter out all operation calls that are not specified in the model. In addition, all calls inside one package (e.g. calls to subroutines inside the application) are omitted.

During the filtering step, a sequence diagram (i.e. *scenario*) is generated for each trace. In the sequence diagram, all method calls occur only between two participants (*Application* and *API*). The actual participant is still saved to be later used when usage scenarios are created. An example of a filtered sequence diagram is shown in Fig. 2.

Creating a filtering model is not always straightforward. Especially generated classes (e.g. *stubs*, *proxies* and *wrappers*) are usually placed into application's package hierarchy, but are logically part of the API. Also, it might be difficult to achieve an optimal size for a filtering model. A too large model will yield to unnecessary operation calls, while missing classes or erroneous categorization may filter out (sometimes even mandatory) interaction patterns.

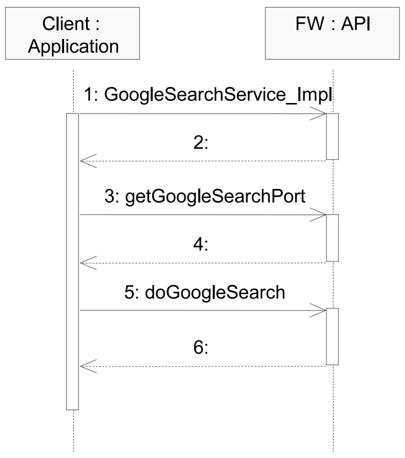

Figure 2. A Filtered Sequence Diagram

3.3 Merging Trace Information

Often, when tracing an application, we get scenarios presenting only a partial viewpoint of the API usage. For example, in a case of interactive programs, user's choices affect the execution flow of the application. To handle variation in API usage, we need to merge the scenarios to get more accurate and complete information (step 3).

To merge the scenarios, we transform the filtered sequence diagrams to a state machine. This transformation

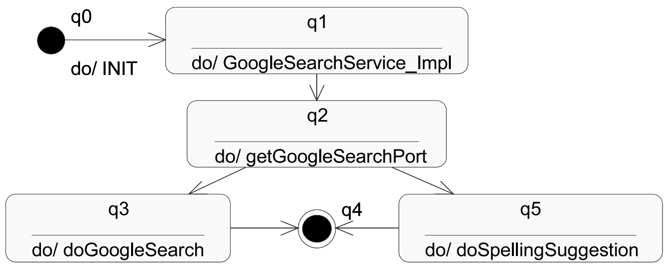

Figure 3. Synthesized State Machine

problem has been widely studied (e.g. [12, 18–20]). For our approach, we use *Minimally Adequate Synthesizer* (MAS) [10, 12]. In essence, MAS maps each method call to an action of a state and each received method call to a transition. The resulting state machine is synthesized based on scenarios representing the usage of the API. Thus it represents the behavior of the API.

The size of the generated state machine depends on the length and similarity of the input scenarios. Without generalizations, every method call is translated to its own state having one outgoing transition (i.e. return of the call). During the generalization process, states calling the same method are merged if possible. Because of the merging, the synthesized state machine will accept execution paths that are not included in the original inputs. Sometimes, in case of *overgeneralization*, some of the new paths may not be wanted.

MAS can be run either in an automatic or in an interactive mode. When run in an automatic mode, MAS forms minimal state machine, thus possibly resulting in overgeneralizations. When run in interactive mode, MAS asks the user if certain generalizations are to be allowed or not in the resulting state machine. We can save the questions (and corresponding answers) and fine-tune the generalization process by modifying the answers. We treat the applications and API as black boxes, thus requiring no prior knowledge on the API invocation orders. Thus, we run MAS in automatic mode.

A state machine synthesized from two traces, the filtered trace in Fig. 2 and another one indicating a call of *doSpellingSuggestion* instead of *doGoogleSearch*, is presented in Fig. 3.

3.4 Model Transformation: From State Machines Back to Sequence Diagrams

In step 4 of the process we generate the usage scenarios using information on the participants involved in the sample interactions and information synthesized in the state machine. The latter is needed e.g. for identifying the common behavioral patterns (including loops) as well as optional and alternative behaviors. These behaviors are translated into corresponding UML2 structured interaction fragments (i.e. *CombinedFragments*).

For our model transformation purposes we utilize one of the widely adopted languages, namely, Atlas Transformation Language (ATL) [5]. The transformation generates a sequence of exchanged messages between lifelines by navigating the state machine diagram and applying predefined transformation rules.

The transformation rules define mappings between state machine and UML2 sequence diagram constructs. Most of the transformation rules are listed in Table 1, where state machine constructs are given on the left column and corresponding UML2 metamodel concepts are listed on the right column. First of all, the transformation generates a *Collaboration*, an *Interaction*, two *Classes* (Application and API), two *Properties* referring to the generated *Classes*, and two *Lifelines* representing the created *Properties*. For each action of state, a sent *Message* and a reply *Message* are created for a synchronous call. A similar transformation process is applied to triggers of transitions.

State Machine	UML2 Sequence Diagram
State Machine	Collaboration, Interaction, Lifeline, Class, Property
Action of State	Message, Operation of Class
Trigger of a transition	Message, Operation of Class
Multiple outgoing transitions from a State	Alternative CombinedFragment (*alt*)
State which are unnecessary to be passed through	Optional CombinedFragment (*opt*)
Loop structure	Loop CombinedFragment (*loop*)

Table 1. Mappings Between State Machine and UML2 Sequence Diagram Constructs

The generated messages are grouped by different kinds of interaction fragments. For instance, consider a situation where there are multiple outgoing transitions from a state, that is to say, the upcoming states are navigated along different paths. Then, the generated events are grouped by alternative fragments to allow one choice for each path.

As the final step, we restore the original API class names saved in the filtering step and expand the API participant with actual names of the called classes.

When generating the usage scenario for the synthesized state machine (Fig. 3) according to transformation rules, actions *GoogleSearchService_Impl* and *getGoogleSearch-*

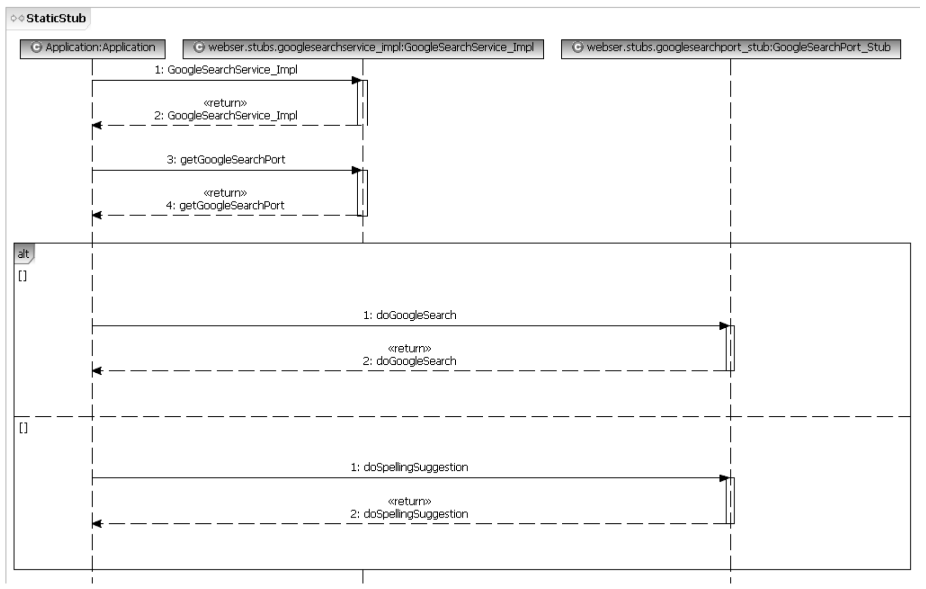

Figure 4. Generated Usage Scenario for the State Machine

Port generate a message sequence. *doGoogleSearch* and *doSpellingSuggestion* actions generate an alternative fragment (shown in Fig. 4).

3.5 Simplifying Usage Scenarios

Even though combined fragments group sequence diagrams in a compact structure, we perceived in practice that a sequence diagram with more than four or five levels of nested combined fragments cannot be easily understood. One of the reasons for nested combined fragments in the generated sequence diagram is strongly connected components (SCC) [1] occuring in state machines. In a SCC, every vertex can be reached from any other vertex inside the SCC. In state machine diagrams, loops connected with each other form a SCC. In the following, SCCs mean connected loops. In some cases, the structures of SCCs can be quite complicated, making the generated sequence diagram less comprehensible. We have decided to disconnect SCCs into several separate loops in a state machine, thus reducing nested layers in the resulting usage scenario.

To disconnect a SCC, in step *i*, a depth-first search to a state machine is performed to derive a depth-first spanning tree, where the edge x→y between states x and y is marked as a *sp-back* edge if y = x or y is an ancestor of x in the spanning tree [17]. *Sp-back* edges result in loops and thus a SCC. In step *ii*, one *sp-back* that connects to more loops than the others is chosen to be removed. If there are several such *sp-back* edges (connecting to the same amount of loops), then the algorithm randomly chooses one of the edges to be removed in step *iii*. Steps *ii* and *iii* are repeated until the state machine diagram no longer contains SCCs.

Removing SCCs from a state machine means that some method calls get lost. We do not see this very problematic for various reasons. First, the SCCs may result due to the overgeneralizations made by MAS, and are thus not present in the example traces. Second, the usage scenarios synthesized are ment to approved and perhaps revised by the analyzer later on. Third, the information on the transitions removed from the state machine is stored and can thus be used later on when revising the usage scenarios.

4 Discussion

When reusing a library, application framework, or software component, it is essential to both identify the available services and understand the correct order these services are to be called to implement a desired functionality. In this paper, we propose an approach to support software reuse and redocumentation of APIs by analyzing the run-time behavior of sample applications. The API usages of the applications are summarized as *usage scenarios*, shown as UML2 sequence diagrams, that illustrate the common, alternative, and optional parts of the usage scenarios.

The analysis of run-time behavior of any, even moderate size software system is very challenging due to the huge amount of monitored trace information. Therefore, we have paid a special attention on pruning and abstracting this information so that only the essential information is presented to the analyzer. In the first phase, we use structural information of the API (filtering model) to focus only on the communication between the application and the API. We further search for behavioral patterns from this pruned trace to identify the common and alternative parts.

While the usage scenarios are useful for comprehending and learning the similarities and differences of API usages of sample applications and for redocumenting and existing API, they can also be used for various other purposes. We have earlier demonstrated their usage to support stepwise implementation of another application using the same API [11]. In essence, the usage scenarios are first transformed into a pattern that presents the structure of sequence diagrams. The patterns can the be applied in a user-assisted way using JavaFrames [7] tool, which proposes the (re)user a partially ordered set of tasks to be carried out when appling the pattern. We have also demonstrated how behavioral rules and requirements captured in a form of UML2 sequence diagrams can be used for validating other applications' API usage [9]. As part of our future work, we intend to integrate the approach proposed in this paper with the methods for utilizing usage scenarios for API usage validation [9] and for step-wise implementation of new applications using the same API [11].

Acknowledgements

This work has been financially supported by Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Solita, Plenware, and Eureka Σ 2023 Programme under grants of the ITEA project if04032 entitled Software Evolution, Refactoring, Improvement of Operational & Usable Systems (SERIOUS). The authors would also like to thank Petri Selonen for his valuable comments.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[2] W. De Pauw and C. Sevitski. Visualizing reference patterns for solving memory leaks in Java. *LNCS*, 1628:116–134, 1999.

[3] R. Delamare, B. Baudry, and Y. L. Traon. Reverse engineering of UML 2.0 sequence diagrams from execution traces. ECOOP'06 Workshop on Object-Oriented Reengineering, 2006.

[4] The Eclipse Foundation, `http://www.eclipse.org/tptp/`. *The Eclipse Test & Performance Tools Platform*, 2006. last visited March 2007.

[5] The Eclipse Foundation, `http://www.eclipse.org/m2m/atl/`. *Atlas Transformation Language*, 2007. last visited March 2007.

[6] Y. G. Gueheneuc and T. Ziadi. Automated reverse engineering of UML v2.0 dynamic models. ECOOP'05 Workshop on Object-Oriented Reengineering, 2005.

[7] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for Java frameworks. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001), Erfurt, Germany, September 2001. LNCS 2186*, pages 163–176, 2001.

[8] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 56–65, Washington, DC, USA, 1997. IEEE Computer Society.

[9] J. Koskinen, M. Kettunen, and T. Systä. Profile-based approach to support comprehension of software behavior. In *Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC)*, pages 212–224, 2006.

[10] J. Koskinen, E. Mäkinen, and T. Systä. Implementing a component-based tool for interactive synthesis of UML statechart diagrams. *Acta Cybernetica*, 15(4):547–565, 2002.

[11] J. Koskinen, A. Ruokonen, and T. Systä. A pattern-based approach to generate code from API usage scenarios. *Nordic Journal of Computing*, 13(2006):162–179.

[12] E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in UML. In *Proceedings of ICSE'01*, pages 15–24. Toronto, Canada, Springer, 2001.

[13] R. Oechsle and T. Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2001.

[14] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. *LNCS*, 2269:151–162, 2002.

[15] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC '06)*, pages 84–88. IEEE Computer Society, June 2006.

[16] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Softw. Pract. Exper.*, 31(4):371–394, 2001.

[17] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[18] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[19] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE*, pages 314–323, 2000.

[20] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *ICSE*, pages 242–251. IEEE Computer Society, IEEE Computer Society, 2004.