IMPROVING MODELS FOR MODEL-BASED TESTING BASED ON EXPLORATORY TESTING

A Thesis

by

Ceren Şahin Gebizli

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
August 2014

IMPROVING MODELS FOR MODEL-BASED TESTING BASED ON EXPLORATORY TESTING

Approved by:

Professor Hasan Sözer (Advisor)
Department of Computer Science
Özyeğin University

Professor Barış Aktemur
Department of Computer Science
Özyeğin University

Professor Ali Özer Ercan
Department of Electrical and Electronics Engineering
Özyeğin University

Date Approved: 2014

ABSTRACT

Model-based testing facilitates automatic generation of test cases by means of models of the system under test. Correctness and completeness of these models determine the effectiveness of the generated test cases. Critical faults can be missed due to omissions in the models, which are primarily created manually. In practice, these faults are usually detected with exploratory testing performed manually by experienced test engineers. In this thesis, we propose an approach for refining system models based on the experience and domain knowledge of these test engineers. Our toolset analyzes the execution traces that are recorded during exploratory testing activities and identifies the omissions in system models. The identified omissions guide the refinement of models to be able to generate more effective test cases. We applied our approach in the context of two industrial case studies to improve the models for model-based testing of a Digital TV system. After applying our approach, three and four critical faults were detected for the first and second case studies, respectively. These faults were not detected by the initial set of test cases and they were also missed during the exploratory testing activities.

ÖZETÇE

Model bazlı test, test edilen sistemin modelleri sayesinde otomatik olarak test senaryoları oluşturulmasını sağlar. Bu modellerin doğruluğu ve tamlığı, oluşturulan test senaryolarının etkinliğini belirler. Genelde manuel olarak oluşturulan modellerdeki eksikliklerden dolayı bazı kritik hatalar tespit edilemeyebilir. Pratikte bu hatalar tecrübeli test mühendisleri tarafından uygulanan araştırma tabanlı testler ile tespit edilebilmektedir. Bu tezde, araştırma tabanlı test aktivitelerinden elde edilen bilgileri kullanarak test senaryosu oluşturulmasında kullanılan modelleri iyileştirmek için bir yöntem ve araç sunuyoruz. Model iyileştirme aktivitelerinin bir araç ile destekleneceği yarı otomatik bir işleyiş öneriyoruz. Bu araç, araştırma tabanlı test aktiviteleri sırasında kayıt altına alınan yazılım koşum yollarını analiz etmekte ve sistem modellerindeki eksiklikleri belirlemektedir. Belirlenen eksiklikler, daha verimli test senaryoları elde edilebilmesi için modelde yapılması gereken iyileştirmeler konusunda bir geribesleme sağlamaktadtır. Bu yaklaşımımızı, bir Dijital TV sisteminin model bazlı testi için model geliştirmesi amacıyla, iki endüstriyel vaka çalışması kapsamında değerlendirdik. Yaklaşımımızı uyguladıktan sonra, birinci endüstriyel vaka çalışmamız sonucunda, 3 kritik hata bulduk. İkinci endüstriyel vaka çalışmamız sonucunda ise, 4 kritik hata daha bulduk. Bu hatalar ilk olarak oluşturulan test senaryoları ile bulunamamıştı ve bu hatalar ayrıca tecrübeye dayalı test aktiviteleri süresince de bulunamamıştı.

ACKNOWLEDGMENTS

We would like to thank software developers and software test engineers at Vestel Electronics for sharing their code base and experiences with us and supporting our case studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

"Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test"

C. Kaner [1]

Software testing is generally considered as a process of executing test cases, which are designed by using test case design techniques. From a broader perspective, it can be defined as any activity aimed at evaluating an attribute or capability of a program/system and determining if it meets its requirements [2].

There exist many different testing methods in software engineering today and there are several test design techniques that are used for designing test cases for software testing [3, 4, 5]. Over the last decade, many of these techniques have been employed in the industry to increase the quality of software systems. Model-based testing (MBT) and exploratory testing can be counted as two of such techniques.

MBT [6] systematizes test case generation based on models that represent the desired behavior of the system under test (SUT) [7]. The effectiveness of the generated test cases relies on the correctness and completeness of these models. Critical faults can be left undetected if the associated scenarios are not reflected to the models. More often than not, this happens to be the case because the models of the SUT are usually defined manually, based on (informal) functional requirements.

Exploratory testing is commonly applied as a complementary approach. Hereby, test engineers make use of their domain knowledge and experience to perform tests and improve their testing strategies by using the new knowledge gained during the testing process [8, 9]. It was showed that exploratory testing is highly effective in practice

to detect critical faults [10] and to improve the effectiveness of the testing process [11]. In practice, however, exploratory testing activities are performed manually and their success depends on the experience and skills of the test engineer. Moreover, the experience gained during these activities is not exploited effectively for improving other techniques such as MBT.

In this work, we aim at utilizing the knowledge and experience gained during exploratory testing activities as a feedback to improve the models of the SUT and as such, generate more effective test cases during the application of MBT. In our approach, we employ event sequence graphs (ESGs) [12] to model system behavior. The execution traces are recorded as a sequence of events observed during the exploratory testing activities. Our toolset analyzes these traces and the ESG model of the SUT to provide a list of warnings regarding missing paths (events and/or transitions among events) in the model. The test engineer can use these warnings to refine the ESG model and generate a new set of test cases.

We applied our approach in the context of two industrial case studies to improve the models for the MBT of a Digital TV system. In the first case study, three critical faults were detected after applying our approach to improve the models of the SUT and the generated test cases. These faults were not detected by the initial set of test cases and they were also missed during the exploratory testing activities. Similarly, we have found four additional faults after applying our approach in the second case study.

The remainder of this thesis is organized as follows. In Chapter 2, we provide background on exploratory testing and MBT, the ESG formalism and our modeling approach. We present the overall approach in Chapter 3. The approach is illustrated in Chapter 4, in the context of the industrial case studies. Hereby, we present and discuss the results as well. Chapter 5 summarizes the related studies. Finally, in Chapter 6, we provide the conclusions and discuss possible future work directions.

CHAPTER II

BACKGROUND

In this chapter, we provide relevant background information regarding exploratory testing and model based testing. The chapter is divided into two sections. First, exploratory testing is described in the following section. Then, we introduce model based testing in the second section. In particular, we explain and illustrate the usage of ESG models, which are employed in our approach and in the case studies.

2.1 Exploratory Testing

Exploratory software testing [3, 2] is characterized by a continuous learning and adaptation process, where the tester iteratively learns about the product and its faults, plans the testing work to be done, designs and executes the tests, and reports the results. The tester dynamically adjusts test goals during execution and prepares only lightweight documentation[13]. The basic approach is depicted in Figure 1.

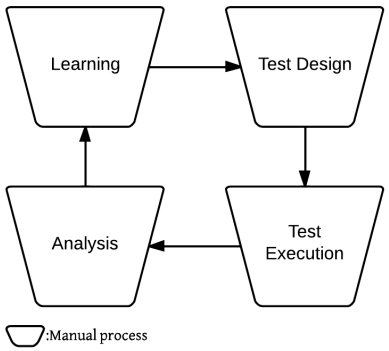


Figure 1: The basic exploratory testing approach.

As the main difference from traditional software testing, exploratory testing is not based on a set of predesigned test cases. Instead of predesigned test cases, testers use their creativity and experiences to steer the process dynamically. Test design,

execution and learning are all concurrent activities in exploratory testing, which aims at reducing the testing cost by utilizing human intuition and experience. It is based on the error guessing technique described in Bach's report [14]. Hereby, Bach briefly defines exploratory testing as simultaneous learning, test design and test execution. There also exist other various definitions [1, 8]. For instance, Tinkham and Kaner define exploratory testing as "*a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project*" [1].

There are no formal descriptions or detailed methodologies defined for exploratory testing yet. Neither there exist strictly described procedures that should be followed by testers during test execution. That is why, exploratory testing has been mainly considered to be an ad-hoc approach. Nevertheless, it is one of the mostly applied and one of the most successful approaches [10], also based on our observations in the industry[1]. Although, formal and automated techniques such as model-based testing are also being applied, critical faults are more often revealed during the exploratory testing activities. This fact also provided the motivation for the problem addressed in this thesis.

2.2 Model-Based Testing

Model-based testing (MBT) is a testing technique that systematizes test case generation based on models that represent the desired behavior of the system under test (SUT) [6, 7]. It has been employed in the industry for more than a decade to increase the effectiveness and efficiency of the testing process and for improving the software quality [15]. The overall MBT process is depicted in Figure 2.

[1]We discuss our observations based on the industrial case studies in Chapter 4.

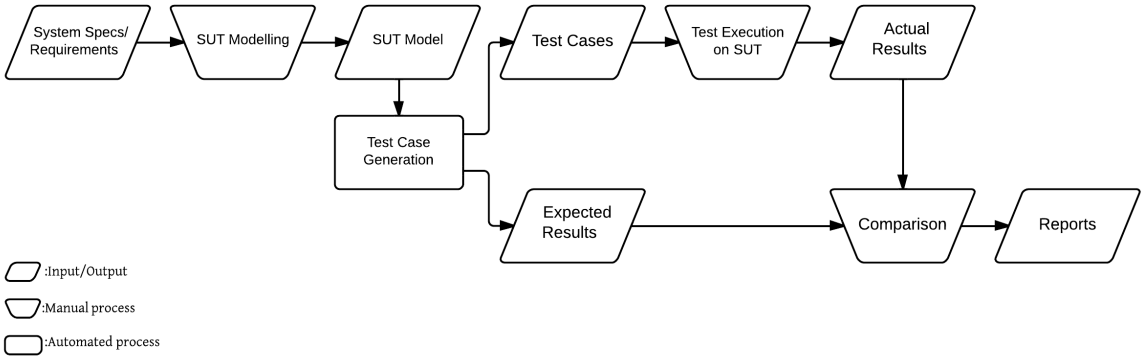


Figure 2: The basic model-based testing approach.

First, system requirements are manually analyzed to create a model of the SUT. This model defines the expected behavior of the SUT with respect to a set of inputs and actions of the user. The SUT model is provided as an input to a MBT tool, which automatically generates a set of test cases by traversing the possible behavioral scenarios on the model. These test cases are executed and compared with respect to the expected results to report any deviation from the expected behavior. In principle, the test case execution and the comparison of results can also be automated. However, MBT is mainly concerned with the automation of test case generation. This automation decreases the testing time and helps to achieve increased (and measured) coverage of possible execution scenarios. In addition, the SUT model and the generated test cases help to document and analyze the system behavior. Changes in requirements can be reflected to the SUT model to generate new test cases with less effort compared to manual test case development.

There are several types of formalisms that are used for expressing the SUT model. These include finite state automaton (FSA), Unified Modeling Language (UML), Markov chains and Event Sequence Graphs (ESG).

FSA is a commonly utilized formalism for MBT, especially to represent state-based behaviors of a system [16]. It is mainly depicted with a set of inputs and states as presented in Figure 3(a). States are represented by nodes and inputs are

annotated on the edges, which represent transitions among the states. FSA is scanned for executable paths to generate test cases. Each possible execution path can be specified as a test case.

UML state charts [17] are similar to FSA models, but they are more complex to comprise additional information associated with a state. They can be used to model the dynamic behavior of classes, use cases, subsystems or the whole system. Conformiq[2] is a MBT tool for embedded software that provides automatic test case generation, execution and analysis based on UML models.

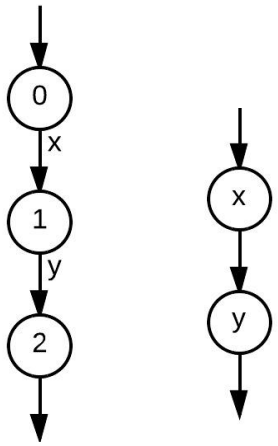
A Markov chain is a discrete-time stochastic process with the Markov property [18]. The main advantage of this formalism is to be able to specify probabilities for state transitions [19]. The system may change its state from the current state to another state, or remain in the same state, according to a probability distribution. Markov Chains are usually used for statistical analysis and reliability assessments. MaTeLo[3] is a MBT tool that employs Markov Chain models as input. It makes use of the specified state transition probabilities for generating test cases that cover the most probable execution scenarios.

In our approach, we used the ESG formalism to express the models of the SUT. This formalism and our modeling approach is described in the following.

2.2.1 ESG Models

In this section, we briefly introduce the ESG formalism and our modeling approach that we used for defining SUT models to generate test cases. ESG facilitates hierarchical modeling and it employs a more abstract representation compared to a state transition diagram or FSA [16]. As such, ESG is a simplified model relative to FSA. Hereby, inputs and states are represented together by assigning them to edges [12] [20]. Each node represents an *event*, which is a user-observable action. Transitions

[2]http://www.conformiq.com
[3]http://www.all4tec.net



(a) FSA (b) ESG

Figure 3: Comparison between a FSA model and the corresponding ESG model.

among the events, i.e., edges, are not labeled.

For example, Figure 3(b) depicts an ESG model that is equivalent to the FSA model presented in Figure 3(a). In Figure 3(a), there exist two transitions labeled as "x" and "y". In the corresponding ESG model, "x" and "y" are represented with nodes in the graph as events.

Figure 4 presents an example ESG model of a simple music player. The basic events are represented as nodes. The model starts with special start node, which is labeled as "$\lfloor$" and ends with a special end node, which is labeled as "$\rfloor$". We can see that one of the nodes (labeled as "*Jump*") is drawn with dashed lines. This notation means that such events further comprise other sub-models.

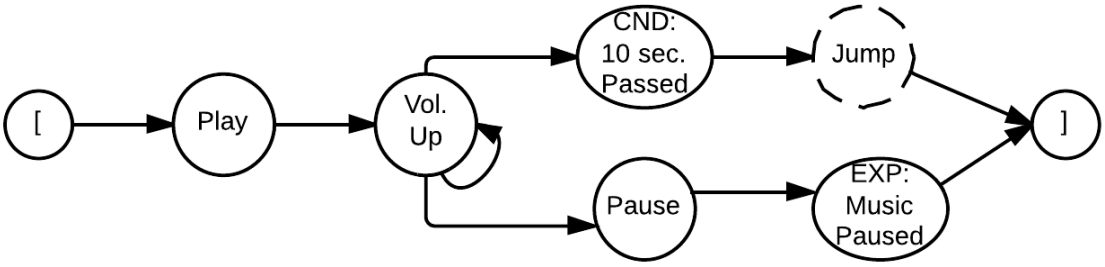


Figure 4: An example ESG model of a simple music player.

In a typical ESG model, events correspond to user-observable actions in general. However, we also needed to explicitly represent some special types of events to facilitate automated testing. For instance, in some cases, a special event is expected to be observed just after the preceding event. In some other cases, a condition must hold before switching to the next event. In our modeling approach, we used the prefixes "*EXP:*" and "*CND:*" for representing *expected events* and *conditions*, respectively. Special scripts are created to check these two types of events and automate the test execution and the comparison of outcome with expected conditions/results. In Figure 4, we can see one condition ("*10 sec. Passed*") and one expected event ("*Music Paused*") depicted as an example.

ESG models are created manually based on informal specifications. Usually the possible actions of the user, events and conditions are not completely and precisely defined in these specifications. As a result, models can be incomplete and/or incorrect with respect to the actually implemented system. In the following section, we describe our approach to overcome such deficiencies by refining ESG models based on exploratory testing.

CHAPTER III

OVERALL APPROACH

The overall approach is composed of three steps, which rely on a set of tools and artifacts as depicted in Figure 5.

The first step is to record test engineer's interaction with the SUT. Hereby, test engineers perform exploratory tests based on their knowledge about the SUT and we collect execution traces with a built-in tool, *Execution Trace Logger*.

In the second step, the collected execution traces are analyzed by our *Model Refinement Evaluator Tool*. The tool takes two more inputs in addition to the execution traces: *i) the existing system model* in ESG format [12], and *ii) the mapping specification*, which defines a mapping from the events taking part in the collected traces, to the events in the model. The mapping is specified in the form of regular expressions. The tool generates a list of warnings regarding the missing paths in the model, which can include a set of unmatched events and/or missing transitions among the events.

The third step is to refine the model according to these warning messages. Once the model is refined, test cases can be generated and executed again. Currently, the first two steps are automated, whereas the third step is performed manually.

In the next chapter we introduce two industrial case studies and illustrate all the steps of our approach in the context of these case studies. We also explain the employed tools, models and techniques in more detail.

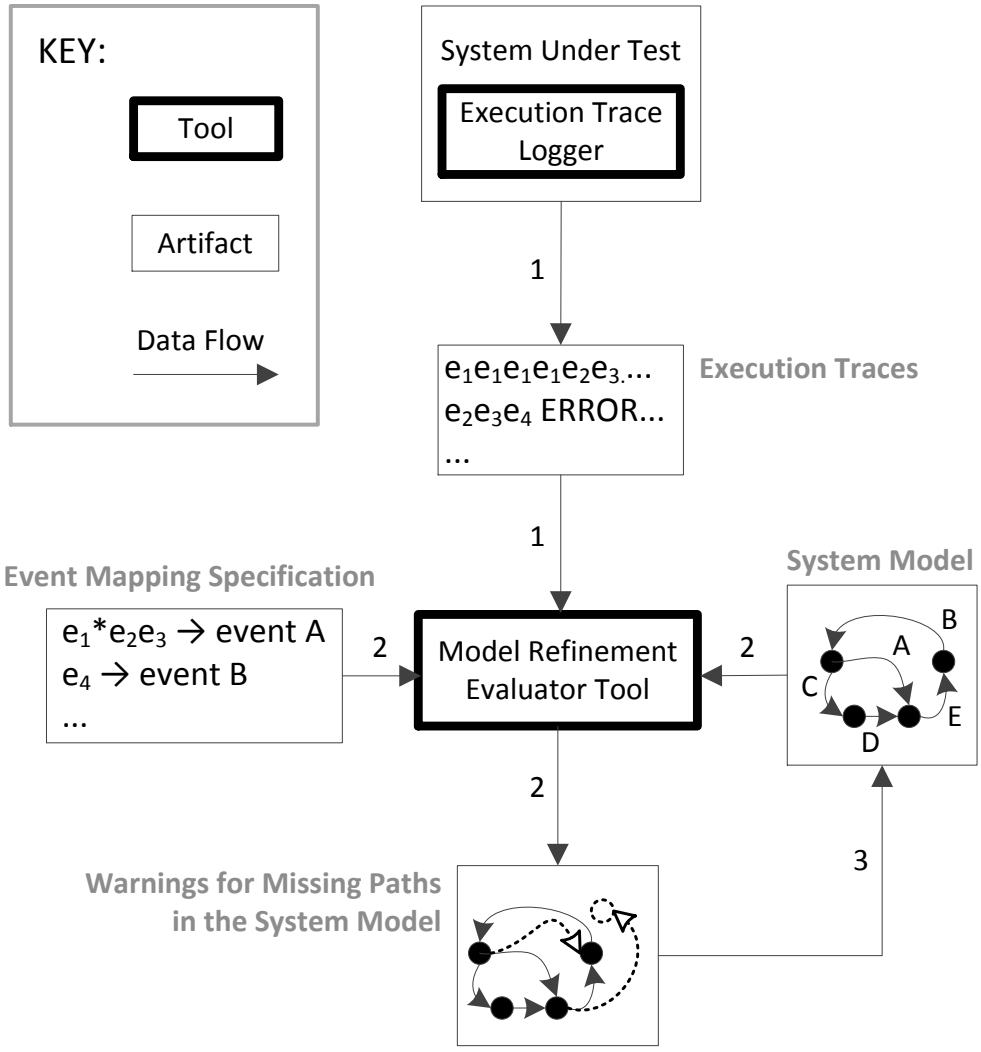


Figure 5: The Overall Approach.

CHAPTER IV

INDUSTRIAL CASE STUDY: DIGITAL TV

TV Platforms are in fast transformation from old electromechanical systems to complicated software systems. Due to complex interconnectivity and aggressive market prospects, one of the biggest challenges becomes system verification. Migration from using traditional testing techniques to advanced, automated testing techniques not only provides effective use of labor force but also shortens the time to market, lowers product price (by increasing the total life time of the product in the market and as such, the total revenue) and improves product quality; these are the uplifting 3 factors that enable market success. Quality expectations are high, whereas resources are limited in the consumer electronics domain. This makes automation essential to be able to detect and remove faults despite limited resources. Therefore, MBT is being adopted to automate test case generation for various features related to different modules of the system. In our studies, we focused on two of such modules in Digital TV. One of them is related to channel installation, named as the Digital Video Broadcasting - Terrestrial Channel Installation (DVB-TCI) module. The other one is the Media Browser (MB) module. These modules are developed and maintained by Vestel[1], which is one of the largest TV manufacturers in Europe.

In the following subsection, we first explain the initial ESG models of the DVB-TCI and MB modules. Then, we explain the mapping of these models to the collected execution traces. Then, we describe the model refinement process. Finally, we present and discuss the results.

[1] http://www.vestel.com.tr

4.1 ESG Models

We designed two ESG models for two case studies. DVB-TCI module of a Digital TV was used for the first case study and Media Browser module was used for the second case study.

4.1.1 ESG Model of the DVB-TCI Module

We used an ESG model of the DVB-TCI module that is created based on its requirement specifications. Figure 6 depicts the top level DVB-TCI model created with Test Suite Designer (TSD) tool [12, 20].

Here, the event *AutoSearch* corresponds to the *automatic channel installation* process, which is modeled by another ESG sub-model as depicted in Figure 7.

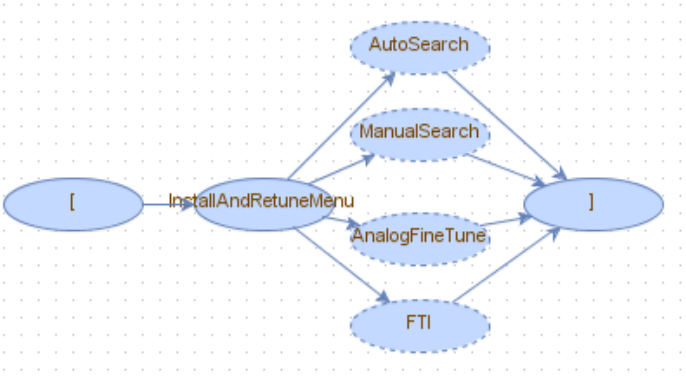


Figure 6: The top level DVB-TCI model.

The TSD tool [20] generated 217 test cases based on the provided ESG models of the DVB-TCI module. TSD is used for creating test cases such that all the paths in the model are covered by limiting the path length by 9. After executing the generated test cases, 3 faults were found. Then, test engineers performed exploratory tests based on their domain knowledge about the DVB-TCI module. We asked them to stop the test when they found a fault and save the records that were logged during tests. The logged data consists of execution traces, i.e., sequence of function calls recorded while

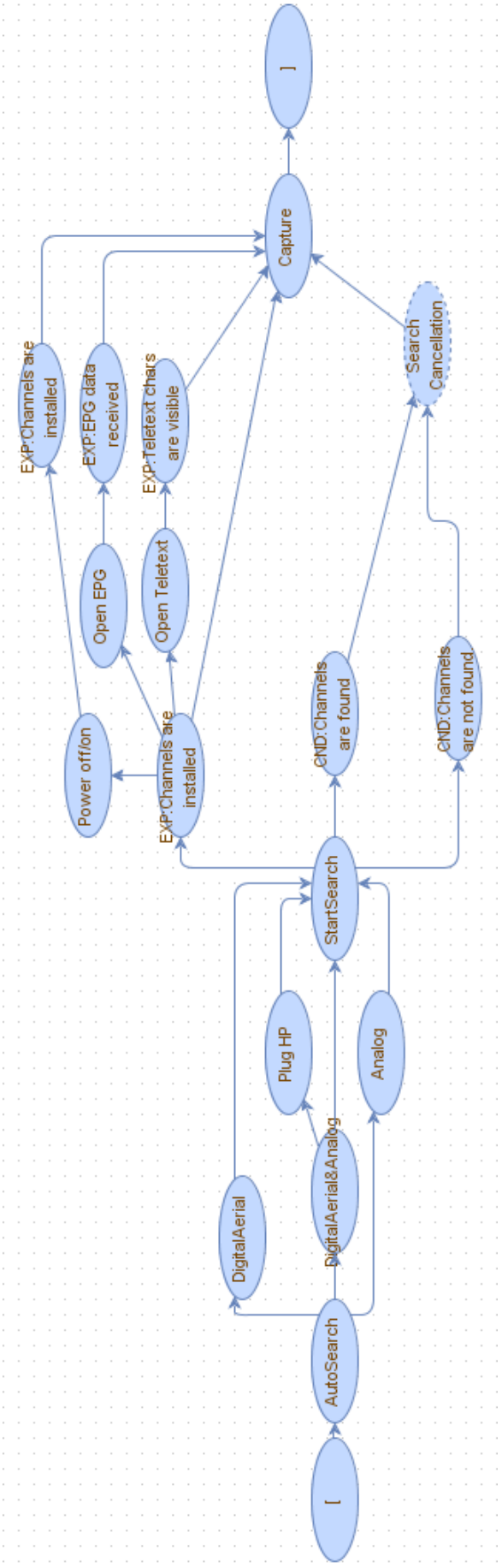


Figure 7: DVB-TCI AutoSearch model.

switching between events [12]. In total 5 faults were found during the exploratory testing activities including the previously found 3 faults.

4.1.2 ESG Model of the MB Module

Figure 8 depicts the top level ESG model of the MB module. Here, the event *Enter MB* corresponds to the *Entering Media Browser* process, which is modeled by another ESG sub-model as depicted in Figure 9.

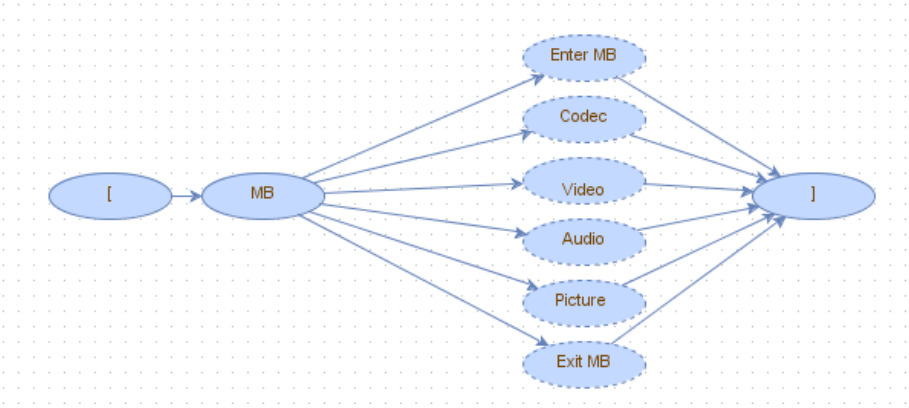


Figure 8: The top level MB model.

The event *Audio* corresponds to the *Audio controls in MB*, which is modeled by another ESG sub-model and this sub-model also includes two further sub-models; *Playing Audio* and *Audio Menu Screen*. *Playing Audio* ESG model is depicted in Figure 10.

The event *Video* corresponds to the *Video controls in MB*, which is modeled by another ESG sub-model and this sub-model also includes two sub-models; *Playing Video* and *Video Menu Screen*. *Playing Video* ESG model is depicted in Figure 11.

The event *Exit MB* corresponds to the *Exiting from Media Browser* process, which is modeled by another ESG sub-model as depicted in Figure 12.

The TSD tool [9] generated 132 test cases based on the provided ESG models of the MB module. TSD is used for creating test cases such that all the paths in the model are covered by limiting the path length by 2. After executing these test cases,

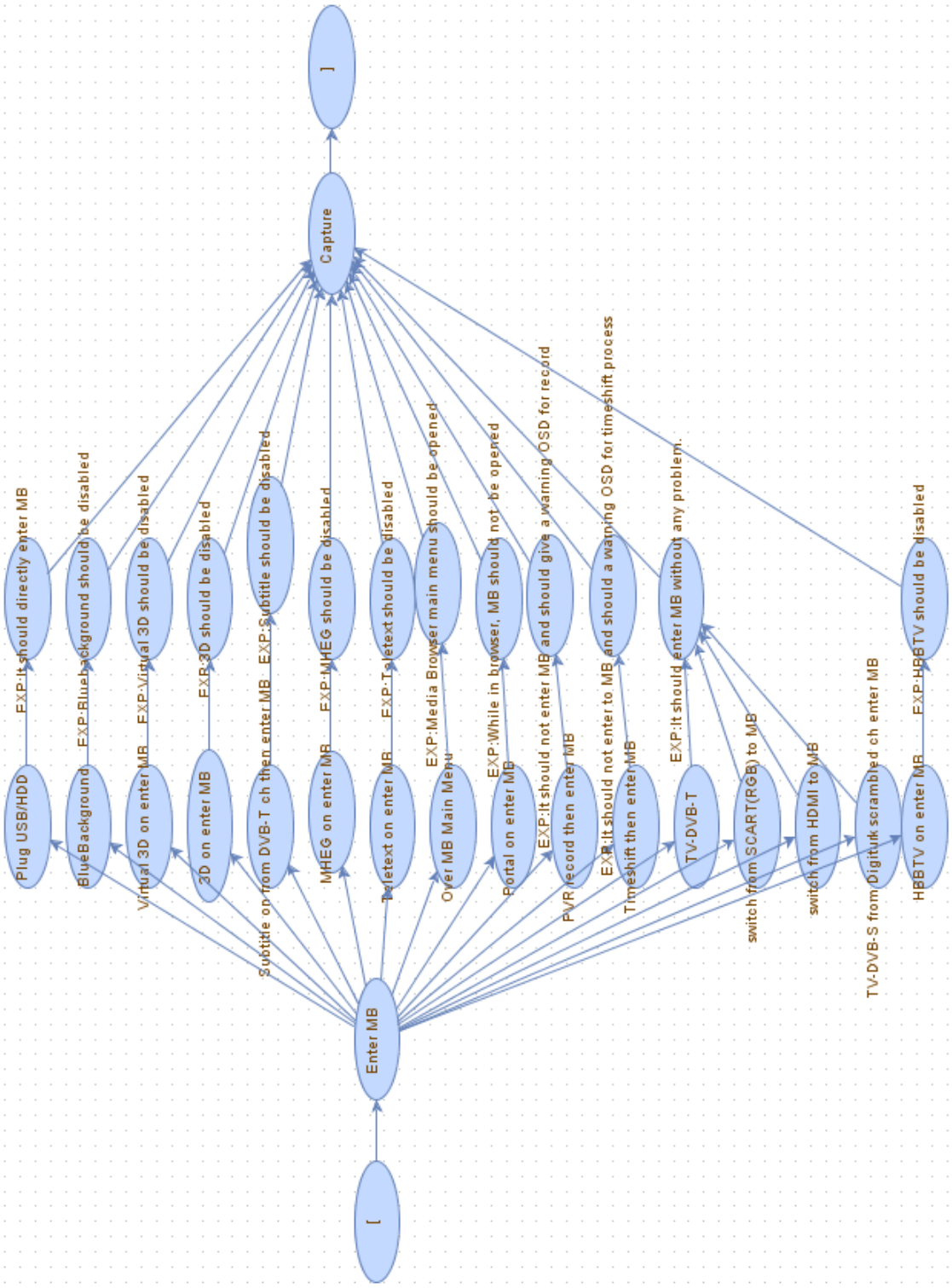


Figure 9: Entering MB model.

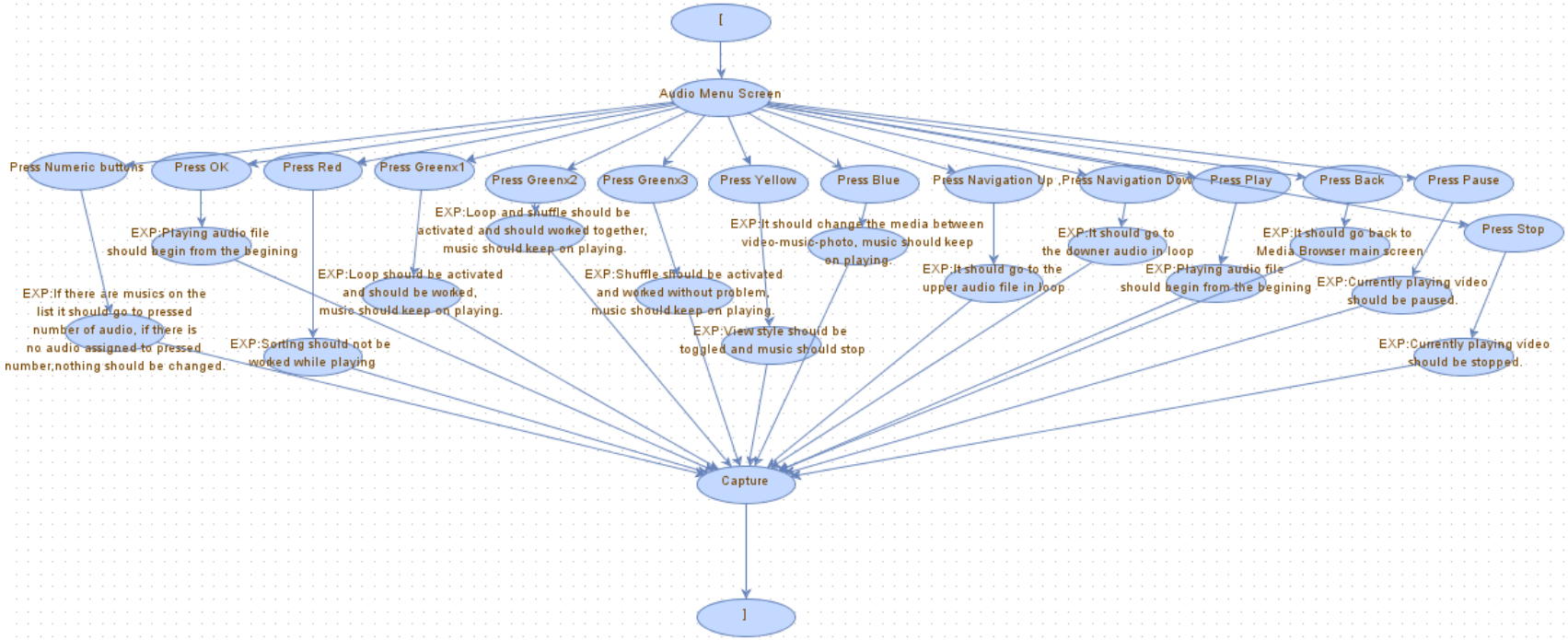


Figure 10: Playing Audio sub-model

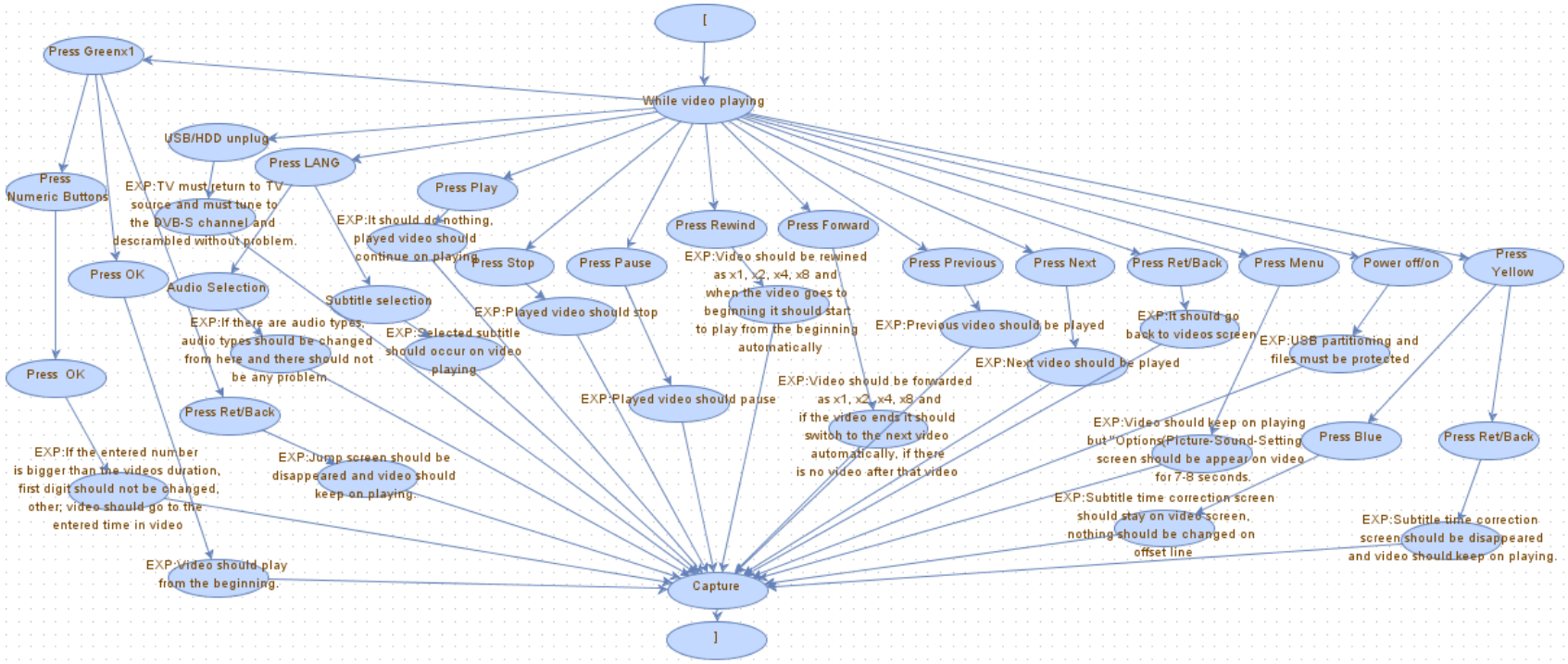


Figure 11: Playing Video sub-model

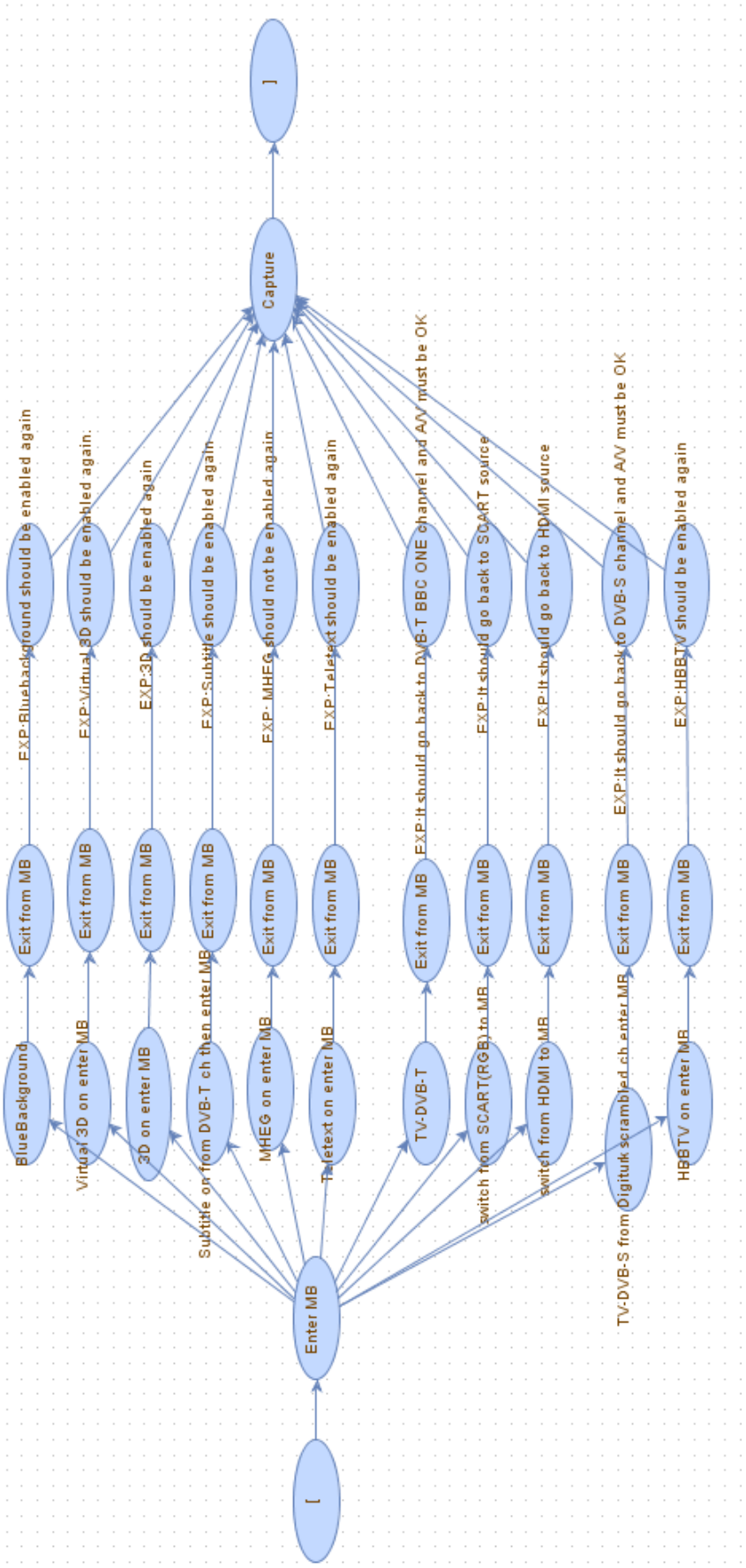


Figure 12: Exiting from MB model.

36 faults were found. Then, test engineers performed exploratory tests based on their domain knowledge about the MB module. We asked them to stop the test when they found a fault and save the records that were logged during tests. In total 44 faults were found during the exploratory testing activities including the previously found 36 faults.

4.2 Mapping Model Elements to Execution Traces

During the exploratory testing activities, *Execution Trace Logger* collects execution traces in the form of sequences of function calls. These function calls usually correspond to the calls that are triggered when the user presses a key on the remote controller. On the other hand, ESG model includes events that are represented at a higher level of abstraction [12]. Therefore, an event mapping specification is used for matching a sequence of recorded function calls to the events or conditions represented in the ESG model.

Listing 4.1 shows a part of the mapping specification created for the DVB-TCI AutoSearch model and Listing 4.2 shows a part of the mapping specification created for the Media Browser model [2].

Listing 4.1: Part of the event mapping specification for the DVB-TCI AutoSearch model.

```
1 // <Event Sequence>     : <Model Element>
2       r                 : Install And Retune;
3       as                : Auto Search;
4       an,s              : Analog;
5       st                : Start Search;
6       ps                : Press Standby;
7       po                : Press 1;
8       f,l*              : Freq=121.00MHz;
```

[2]Due to confidentiality, we do not disclose the real function names used in the implementation.

```
9       m,p*                : Media Video Playing;
10  //
```

Listing 4.2: Part of the event mapping specification for the Media Browser model.

```
1   // <Event Sequence>     : <Model Element>
2       t,s*                : Tune to DVB-S channel;
3       mb                  : Media Browser menu;
4       vd                  : Video part of Media Browser menu;
5       n*,p                : While video playing ;
6       sub                 : Press Subtitle;
7   //
```

Hereby, each event sequence is represented by a regular expression. These expressions are defined manually to map a sequence of low-level events to the events/states in the model. For instance, we can see at line 9 of Listing 4.1 that a function call named *m*, followed by zero or more function calls named *p* notifies that the event *Media Browser Video Playing* has occurred. There is usually a specific function call that handles a remote controller command. For this reason, a single function call is mapped to an event in most cases. Also note that *expected events* and *conditions* are not considered for mapping. Special scripts are created for these events, which take the role of test oracles. Therefore, if an event is labeled with a prefix "*EXP:*" or "*CND:*", it is associated with a test script. Otherwise, the event has to be mapped to a sequence of functions calls in the execution trace.

We developed the *Model Refinement Evaluator Tool* that compares the execution traces of a module, which is collected during exploratory tests, with the ESG model of that module. The tool takes the collected execution traces, the ESG model and the mapping specification as input. It creates a finite automaton for each regular expression. The list of execution traces is treated as a list of strings to check step-by-step if they are accepted by any automaton or not. If a proceeding part of the

execution trace is not accepted by any automaton, it can not be mapped to an event in the ESG model. A warning message is generated in this case regarding a missing event.

The *Model Refinement Evaluator Tool* also checks the conformance of the ESG model with respect to the transitions observed among the events. A warning message is generated if a possible transition among the events is not reflected to the model. For instance, a function call sequence *as, an, s, as* corresponds to the event sequence *AutoSearch, Analog, AutoSearch* according to Listing 4.1. However, there is no edge from *Analog* back to *AutoSearch* according to the ESG model. For the MB case study, a function call sequence *t,s*, mb, vd, n*,p, sub* corresponds to the event sequence *Tune to DVB-S channel, Media Browser menu, Video part of Media Browser menu, While video playing, Press Subtitle* according to Listing 4.2. But there is no edge from *While video playing* to *Press Subtitle* according to the ESG model. Hence, these recorded function call sequences would lead to a warning regarding a missing edge in the model. In the following subsection, we present and discuss the warning messages generated for the case studies.

4.3 Refinement of the Models

4.3.1 Refinement of the DVB-TCI ESG Model

The output of the tool for DVB-TCI case study can be seen in Listing 4.3. There are two warning messages. These warnings are actually related to the faults that were missed by the 217 test cases, but later detected by the test engineer manually. The first warning (Lines 1-5) is regarding a missing event in the model. A function call sequence, which involves multiple calls of the *prgup* function is not mapped to any event in the mapping specification. Based on this warning, we defined a new event, namely "*program up multiple times*", that represents one or more calls to the *prgup* function following each other, i.e., "*prgup+*". We also added a succeeding event that

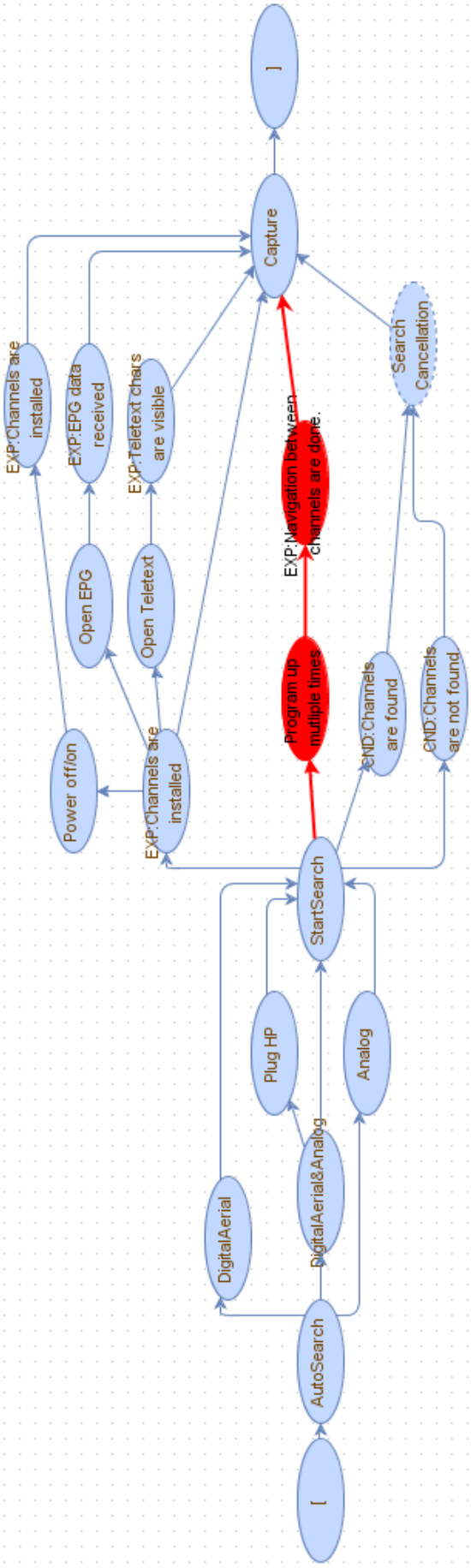


Figure 13: The refined AutoSearch model.

represents the expected event just after the "*program up multiple times*" event (See Figure 13). The second warning (Lines 7-11) is regarding a missing transition from one event to another. We added the corresponding transition to the ESG model as well (See Figure 14).

Listing 4.3: The generated warning messages by the Model Refinement Evaluator Tool for the DVB-TCI ESG model.

```
1  e4: Start search --> prgup, prgup
2  Missing destination event!
3  A new mapping must be added for
4  "prgup, prgup"
5  function call sequence
6
7  e4: Start search --> e1: InstallAndRetuneMenu
8  Missing edge!
9  A new edge must be added
10 from "e4: Start search"
11 to "e1: InstallAndRetuneMenu"
```

In total, we added two new events and four new transitions to the existing ESG models based on the warning messages. Modifications are marked on the refined models depicted in Figure 13 and Figure 14.

4.3.2 Refinement of the MB ESG Model

The output of the tool for Media Browser case study can be seen in Listing 4.4. There are 9 warning messages. These warnings are actually related to the faults that were missed by the 132 test cases, but later detected by the test engineer manually.

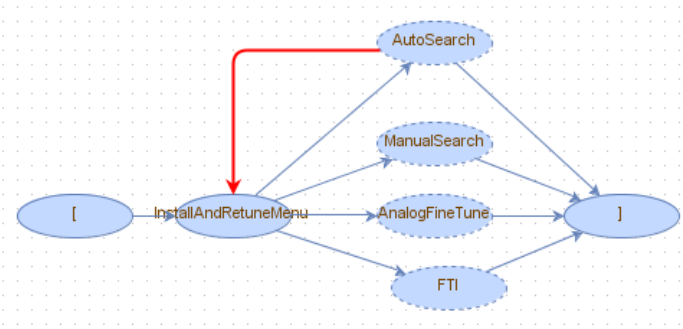


Figure 14: The refined top level DVB-TCI model.

Listing 4.4: The generated warning messages by the Model Refinement Evaluator Tool.

```
1  Tune to analog channel --> e: Enter Media Browser
2  Missing source event!
3  A new mapping must be added for
4  "Tune to analog channel"
5  function call sequence
6
7  e58: While video playing --> Press subtitle for embedded subtitle
8  Missing destination event!
9  A new mapping must be added for
10 "Press subtitle for embedded subtitle"
11 function call sequence
12
13 Do not plug any USB or HDD --> e: Enter Media Browser
14 Missing source event!
15 A new mapping must be added for
16 "Do not plug any USB or HDD"
17 function call sequence
18
19 e58: While video playing --> Switch to SCART source
20 Missing destination event!
```

```
A new mapping must be added for
"Switch to SCART source"
function call sequence

Unplug signal cable --> e: Enter Media Browser
Missing source event!
A new mapping must be added for
"Unplug signal cable"
function call sequence

e63: Press Menu --> Enter picture menu
Missing destination event!
A new mapping must be added for
"Enter picture menu"
function call sequence

Enter picture menu --> e63: Press Menu
Missing source event!
A new mapping must be added for
"Enter picture menu"
function call sequence

e63: Press Menu --> e63: Press Menu
A new edge must be added
from "e63: Press Menu"
to "e63: Press Menu"

e63: Press Menu --> e: Enter Media Browser
A new edge must be added
from "e63: Press Menu"
to "e: Enter Media Browser"
```

In total, we added 23 new events and 28 new transitions to the existing ESG models based on the warning messages. The refined models are depicted in Figure 15, Figure 16, Figure 17 and Figure 18, where modifications are marked.

4.4 Results and Discussions

In the following, we present and discuss the results obtained by applying our approach on the two case studies.

4.4.1 DVB-TCI Module

In total 349 test cases were generated after the refinement of the model. 8 faults were found when these test cases were executed. 5 of these faults had been already detected before, but 3 additional faults were found after employing the refined model. Moreover, these faults were highly critical; one of them caused the TV to reset itself after a channel search is performed; another one was the reason for duplicate channels added at the end of channel search; the activation of the third fault resulted in a crash of the system. Table 1 lists the properties of the model of the DVB-TCI module and the number of faults found before and after the refinement of the model by applying our approach.

Model	# of Nodes	# of Edges	# of Test Cases	# of Faults
Initial	1225	1501	217	3
Refined	2012	2868	349	8

Table 1: Properties of the DVB-TCI ESG model and the number of faults found before and after the refinement of the model based on 2 additional faults found during exploratory testing.

4.4.2 MB Module

In total 139 test cases were generated after the refinement of the model. 48 faults were found when these test cases were executed. 44 of these faults had been already detected before,but we discovered 4 additional faults. These faults were highly critical

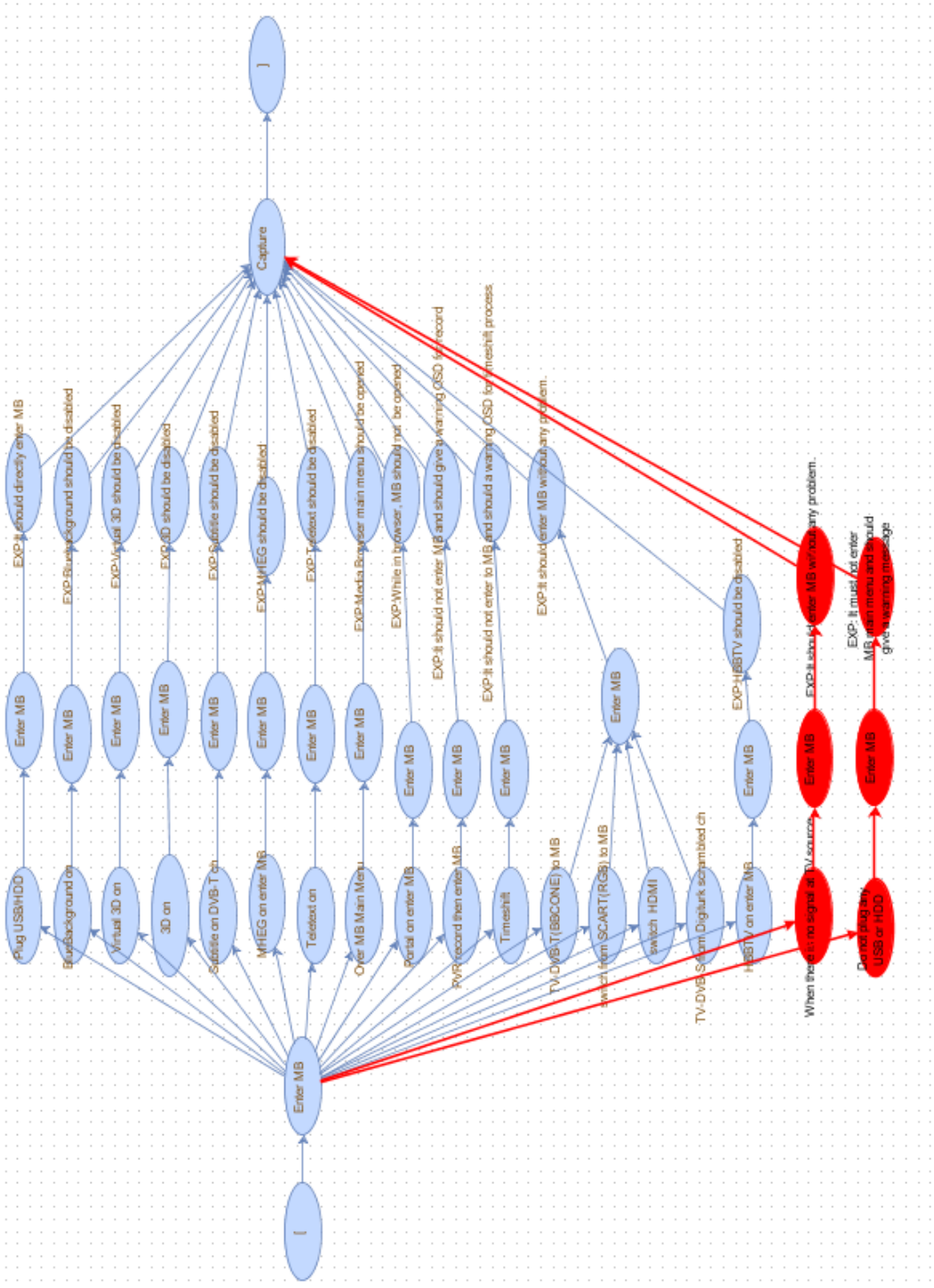


Figure 15: The refined Enter Media Browser model.

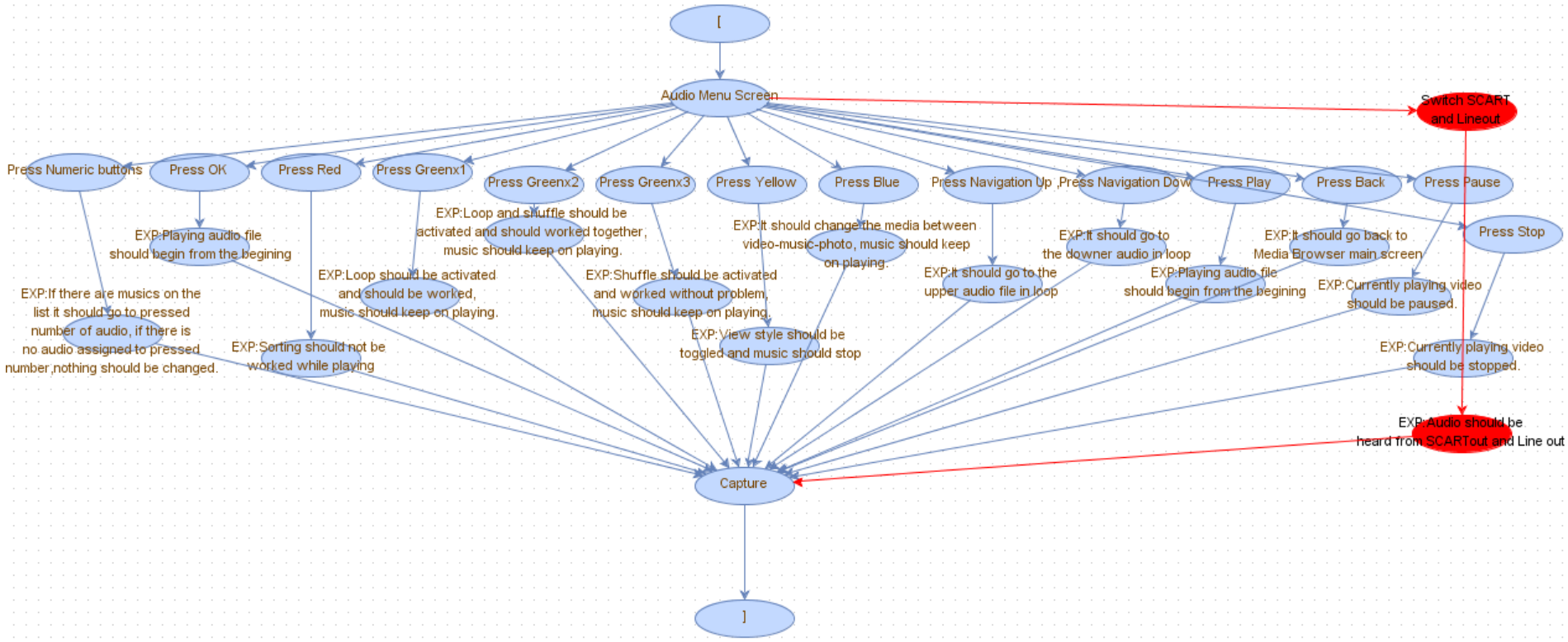


Figure 16: The refined Playing Audio model.

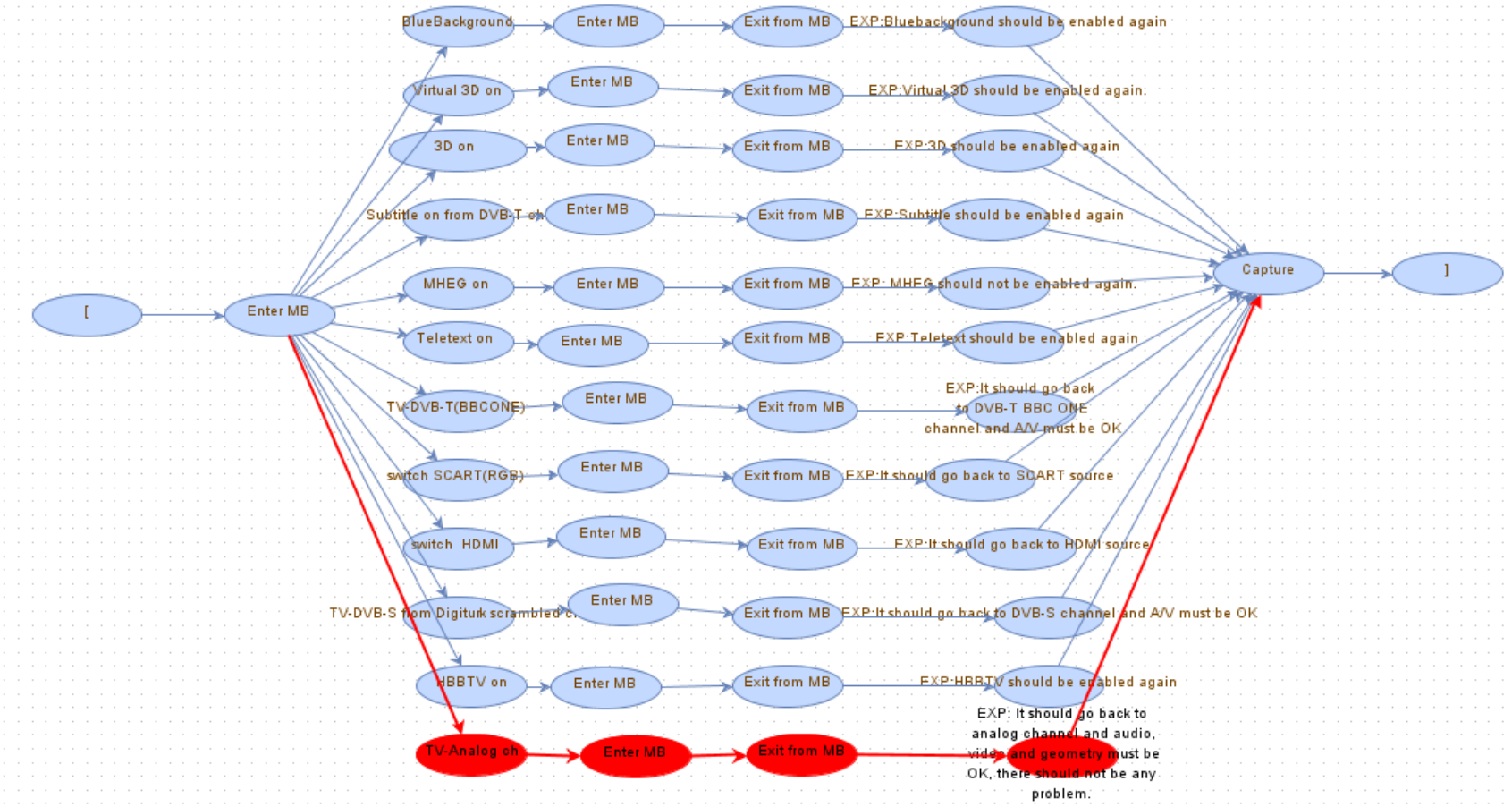


Figure 17: The refined Exit Media Browser model.

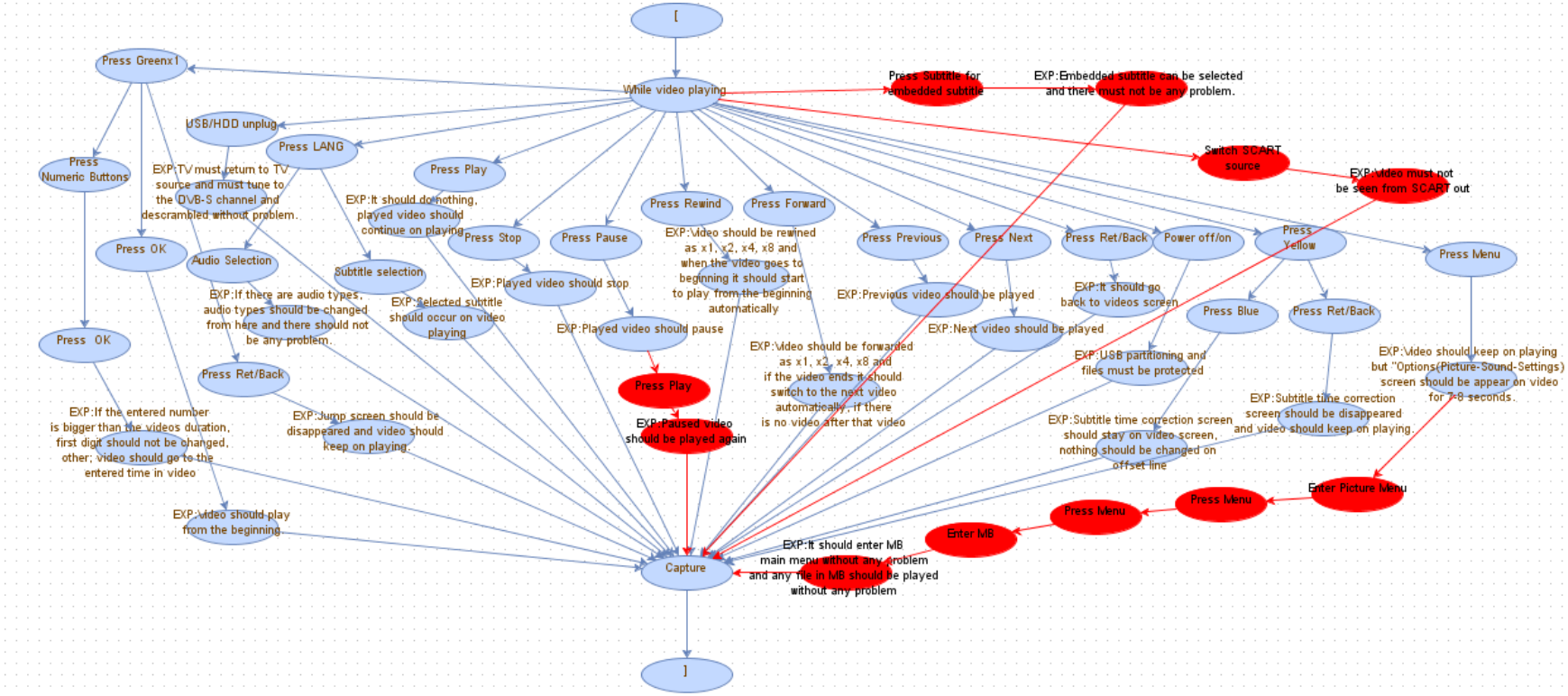


Figure 18: The refined Playing Video model.

as well; the first one made the audio/video output disappear; the second one corrupted remote controller key buffers; the third one made user commands undetectable; the last one caused the TV to reset itself. Table 2 lists the properties of the model of the MB module and the number of faults found before and after the refinement of the model by applying our approach.

Model	# of Nodes	# of Edges	# of Test Cases	# of Faults
Initial	294	424	132	36
Refined	341	478	139	48

Table 2: Properties of the MB ESG model and the number of faults found before and after the refinement of the model based on 4 additional faults found during exploratory testing.

In Table 2, note that the number of test cases increases by 7; however the number of found faults increases by 12. This is because, more than one fault can be found with one test case. In Table 2, note that the number of test cases increases by 7; however the number of found faults increases by 12. This is because, more than one fault can be found with one test case.

4.4.3 Discussions

Results of the case studies show that we can identify critical faults by updating the models based on exploratory testing with the help of the *Model Refinement Evaluator Tool*. Our approach and the tool support enabled the transfer of knowledge from manual exploratory testing activities to MBT.

In the first case study for the DVB-TCI module, number of test cases increased by 60.82% after refinement. The reason for the high increase in the number of test cases is a missing edge update, which caused a loop in a model (See Figure 14). Due to this loop, we can get more test paths and detect more faults, which could not be detected before. As a result, we can find additional critical faults. The number of detected faults is increased from 3 to 8.

In the second case study for the MB module, the number of test cases increased by 5.3% after the refinement. The number of detected faults is increased from 36 to 48. These results were particularly interesting because a high number of critical faults were detected despite of a relatively low number of additional test cases. Refinements focusing on critical parts of the model helped to find other hidden faults that were not detected before.

We observed that the effectiveness of the models used for MBT can be significantly increased. Combining two testing techniques provides us more relevant test paths to explore. We showed that our approach helps to detect critical faults and increase the efficiency of the testing process.

CHAPTER V

RELATED WORK

There exist an extensive literature on model based testing (MBT) approaches [21], tools [22] and different types of models employed like finite state machines [23] [24] and Markov chains [19]. There also exist surveys on MBT [21] and several case studies [22], [25] where the effectiveness of MBT is evaluated. The surveyed studies and the applied MBT techniques are mainly focusing on the modeling approach and test case generation methods. The evaluation is performed by comparing the number of faults detected with respect to the number of faults exposed by traditional testing approaches. Iterative refinement of the developed models and the incorporation of domain knowledge have not taken much attention.

In a recently proposed approach [26], MBT and capture-replay testing techniques are integrated. There are two differences between this approach and our approach. First, the proposed approach focus on refining and adapting a capture-replay testing tool based on changes in the user interface (model). In our work, we focus on refining system models based on the captured execution traces instead. Although we did not consider evolution explicitly, changes in the system can be reflected to our approach in a modular way. In case the system evolves, one only needs to update the event mapping specification. The second difference is that the effectiveness of the proposed approach is not evaluated with case studies or controlled experiments. In this thesis, we provide promising results based on industrial case studies as well.

There exist another approach [27] that proposes the automated refinement of testing models. The approach employs event-sequence based test models for testing applications through their GUI. Relationships among the GUI elements are observed

at runtime, during the execution of the generated test cases. The model is iteratively updated based on the inferred relationships. The main advantage of this approach is that it is fully automated, except the initial creation of the model. The disadvantage is that the approach is limited in terms of potential for improvement. If some part of the GUI is never exercised, for instance, relationships among the elements in that part can never be discovered. Hence, the model can not be updated to take these elements into account.

Exploratory testing has also been evaluated in the context of industrial case studies [28]. Research work in this area include the investigation of the impact of human personality [29], learning styles [30] and the way that different types of knowledge [31] are utilized like domain knowledge, system knowledge and general software engineering knowledge. These studies are mainly based on interviews that are performed at different companies [28].

Exploratory modeling [32] has been introduced as an approach for model development based on the principles of exploratory testing. Hereby, a state diagram of the expected behavior of the SUT is developed first. Then, this model is refined by observing the different states and behaviors while interacting with the SUT. However, this approach has no tool support and it was not evaluated in the context of an industrial case study.

In practice, exploratory testing is usually performed manually and model based testing is applied as a complementary, rather than an integrated approach. To the best of our knowledge, there has been no method or toolset to couple these approaches and utilize the knowledge gained in exploratory testing as a feedback for model based testing. In this work, we showed that such an approach is viable and effective in detecting faults.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

We introduced an approach and a toolset for incorporating information from manual exploratory testing activities into model based testing to increase the effectiveness of models that are used for test case generation. In our approach, the execution traces that are recorded during exploratory testing activities are utilized as a feedback for refining system models. We applied our approach in the context of two industrial case studies to improve the models for model-based testing of a Digital TV system. We could detect real and critical faults. These faults were not detected by model based testing based on existing models and they were also missed during the exploratory testing activities.

In our future work, we plan to extend our toolset to semi-automatically incorporate refinements into existing models. We also plan to conduct more case studies.

References

[1] C. Kaner, "Exploratory testing," in *Quality Assurance Institute Worldwide Annual Software Testing Conference*, 2006.

[2] W. C. Hetzel and B. Hetzel, *The Complete Guide to Software Testing*. New York, NY, USA: John Wiley & Sons, Inc., 2nd ed., 1991.

[3] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[4] C. Kaner, J. L. Falk, and H. Q. Nguyen, *Testing Computer Software, Second Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2nd ed., 1999.

[5] L. Copeland, *A Practitioner's Guide to Software Test Design*. Norwood, MA, USA: Artech House, Inc., 2003.

[6] L. Apfelbaum and J. Doyle, "Model-based testing," in *Software Quality Week Conference*, pp. 296–300, 1997.

[7] J. Boberg, "Early fault detection with model-based testing," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, pp. 9–20, 2008.

[8] A. Tinkham and C. Kaner, "Exploring exploratory testing," in *Proceedings of the Software Testing and Analysis and Review East Conference*, 2003.

[9] C. Agruss and B. Johnson, "Ad hoc software testing: A perspective on exploration and improvisation," in *Florida Institute of Technology*, pp. 68–69, 2000.

[10] J. Itkonen, *Empirical Studies on Exploratory Software Testing*. Ph.D. thesis, Aalto University, 2011.

[11] J. Itkonen, M. V. Mantyla, and C. Lassenius, "Defect detection efficiency: Test case based vs. exploratory testing," in *First International Symposium on Empirical Software Engineering and Measurement*, pp. 61–70, IEEE Computer Society, 2007.

[12] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings of 12th International Symposium on Software Reliability Engineering, ISSRE2001*, pp. 34–43, 2001.

[13] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, 1st ed., 2009.

[14] J. Bach, "Exploratory testing explained," tech. rep., 2003.

[15] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[16] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering 4*, vol. 4, no. 3, pp. 178–187, 1978.

[17] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[18] G. H. Walton and J. H. Poore, "Generating transition probabilities to support model-based software testing.," *Softw., Pract. Exper.*, vol. 30, no. 10, pp. 1095–1106, 2000.

[19] J. Whittaker and M. Thomason, "A markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[20] F. Belli, A. T. Endo, M. Linschulte, and A. Simao, "A holistic approach to model-based testing of web service compositions," *Software: Practice and Experience*, vol. 44, no. 2, pp. 201–234, 2014.

[21] A. C. D. Neto, R.Subramanyan, M.Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pp. 31–36, 2007.

[22] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the International Conference on Software Engineering*, pp. 285–294, 1999.

[23] H. Robinson, "Finite state model-based testing on a shoestring," in *Proceedings of the Software Testing and Analysis and Review West Conference*, 1999.

[24] A. Chander, D. Dhurjati, S. Koushik, and Y. Dachuan, "Optimal test input sequence generation for finite state models and pushdown systems," in *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation*, pp. 140–149, 2011.

[25] J. Keranen and T. Raty, "Model-based testing of embedded systems in hardware in the loop environment," *IET Software*, vol. 6, no. 4, pp. 364–376, 2011.

[26] V. Entin, M. Winder, B. Zhang, and S. Christmann, "Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp. 572–577, March 2011.

[27] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using gui runtime state feedback," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 81–95, Jan. 2010.

[28] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *Proceedings of International Symposium on Empirical Software Engineering*, pp. 84–93, 2005.

[29] L. Shoaib, A. Nadeem, and A. Akbar, "An empirical evaluation of the influence of human personality on exploratory software testing," *Multitopic Conference and IEEE 13th International*, pp. 1–6, 2009.

[30] A. Tinkham and C. Kaner, "Learning styles and exploratory testing," in *Proceedings of the Pacific Northwest Software Quality Conference*, 2003.

[31] J. Itkonen, M. V. Mantyla, and C. Lassenius, "The role of the testers knowledge in exploratory software testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 707–724, 2013.

[32] H. Robinson, "Intelligent test automation a model-based method for generating tests from a description of an applications behavior," *Software Testing and Quality Engineering Magazine*, pp. 24–32, 2000.