2009 Fourth International Conference on Software Engineering Advances

Program Comprehension for User-Assisted Test Oracle Generation

Teemu Kanstrén
VTT Technical Research Centre of Finland
Kaitoväylä 1, 90571 Oulu, Finland
e-mail: teemu.kanstren@vtt.fi

***Abstract*—Software testing requires a test oracle that makes an assessment of the correctness of the tested program behaviour, based on a priori created model. While test automation is a popular research topic, there is only a limited amount of work in the subject of automating the process of creating test oracles. This lack of test oracle automation greatly limits the usefulness of automated testing techniques. One reason for this is the difficulty to automatically determine the correctness of previously unknown software. Instead the task of coming up with a useful oracle is often left to the user as a manual task. Program comprehension techniques are focused on supporting the building of human understanding for a previously unknown program, and as such are good candidates to assist in the test oracle creation process. This paper addresses the lack of automated support for test oracle creation by providing a framework for using program comprehension techniques to provide automated assistance to the user in creating test oracles. Based on analysis of existing work and theoretical background, the basic concept for this process is defined. A case example demonstrates the practical application of this concept with the generation of a model, including a test oracle, for model-based testing. From the existing approaches and the presented case example, a framework for this type of process is presented in order to provide a basis for providing more powerful techniques for user-assisted test oracle generation.**

Keywords- Test oracle; Program comprehension; Test automation

I. INTRODUCTION

Test automation in software engineering often does not live up to its name and promise. Commonly the test automation is actually a set of test scripts written manually and executed over and over by a tool designed for this purpose. This is useful for regression testing but does not deliver on the promise of automated testing, where one could just run a tool to generate tests for a given piece of software without having to manually create them. A truly automated testing platform would need to automatically generate message sequences to drive the system under test (SUT) through its interfaces, test input data for these messages, a test harness to isolate the SUT from its environment and a test oracle to verify the correctness of the SUT output in response to the input messages and data.

With statements on how software testing takes more than 50% of the total development costs [1], test automation has of course been a popular research topic and numerous research papers have been published related to the automation of different test automation components. Especially test input generation has been a popular research area. One of the least automated parts of test automation remains to be the creation of test oracles. This can be seen to be partly due to the difficulties to automatically (like a magical oracle) determine the correctness of previously unknown software. The specification of what is to be expected of a SUT comes from its stakeholders, and no program can guess what is expected from another program without external input. Some generic properties of the correct functionality can be devised for specific cases and domains (e.g. refactoring engines [2] and protocols errors in web applications [3]), but the truly automated parts of these are limited and do not generalize.

This paper views the automatically assisted oracle creation problem as an application of program comprehension (PC). Similar topics have been considered before, for example, Sneed has discussed how the human tester is the person who needs to have the best understanding of the SUT [4]. PC is aimed at building a human understanding of software (SW) systems. Research in this field can be classified to study either the human view of cognitive processes used to understand programs or the technological view of building semi-automated tool support for program comprehension [5]. The end result is typically a model describing the program at a chosen abstraction level and from a chosen viewpoint. Finally this model needs to be validated to ensure correct understanding. This is closely related to how a test oracle works, by comparing a model of the expected SUT behaviour against a model of the actual SUT behaviour and verifying that they match.

The focus of this paper is on creating test oracles for existing systems, with the help of dynamic analysis techniques. In the spirit of PC, the goal is not to achieve fully automated generation of test oracles for any SUT but to provide a framework for how automated assistance for creating the test oracles can be provided for the human user. Starting with a theoretical analysis of the concepts, existing approaches for the subjects are reviewed. Next, an example case of applying PC concepts for the generation of a model, including a test oracle, for model-based testing is presented. Finally, existing approaches are summed up together in respect to the presented theoretical background to provide a framework for providing techniques to support test oracle automation with the help of PC techniques.

The rest of the paper is structured as follows. Section 2 provides a general overview of the test oracle concept and existing techniques to support test oracle automation. Section 3 presents the general concepts of PC, and a brief overview of related techniques. Section 4 provides a model describing

978-0-7695-3777-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ICSEA.2009.26

the relationship between test oracles and PC, including a comparison of the test oracle and PC techniques presented in the previous sections. Section 5 presents an example case of a PC approach to provide automated assistance for creating test oracles for an existing system. Section 6 discusses the case study and its relation to existing work shown in section 4, analyzing these different concepts and presenting a framework for user-assisted test oracle generation with the help of PC techniques. Finally, conclusions end the paper.

II. TEST ORACLES

This section presents an overview on test oracle concepts and related research to provide a basis for analyzing the synergies to program comprehension in later sections.

A. General Concepts

The concepts of test oracles in this paper follow the definitions used in [6]. A *test oracle* is defined as a mechanism for determining the correctness of the behaviour of software during (test) execution. The oracle is divided into the *oracle information*, specifying what constitutes the correct behaviour, and the *oracle procedure*, which is the algorithm verifying the test results against the oracle information. Further terms are also used according to [6]. Successful test evaluation requires capturing information about the running system using a *test monitor*. For simple systems, it can be enough for the test monitor to just capture the output of the system. For more complex systems, such as reactive systems, more detailed information, such as internal events, timing information, stimuli and responses, need to be captured. All the information captured by the test monitor is called the *execution profile* (EP) of the system, and includes control and data information.

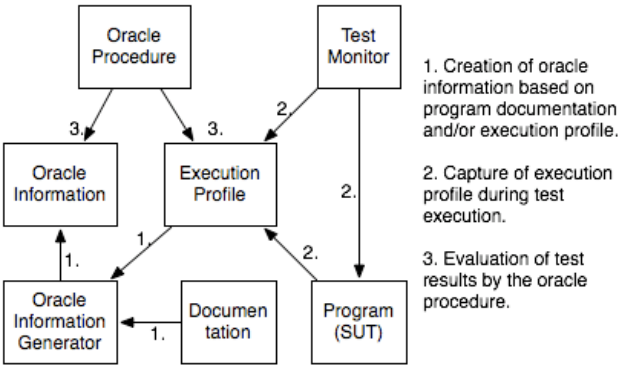

Figure 1. Test oracle components.

The different components of test oracles and their relations are illustrated in Figure 1. These are grouped to three main steps based on the order of their application. Before a test case can be created, the oracle information and procedure need to be defined, which forms the first step. This is typically based on the program specifications and/or EP (in which case step 2 would precede step 1). This information can be generated by a human developer/tester, a test automation program or a combination of both (program supporting a human). In the second step, the test case is executed and the test monitor captures the EP of the system. The input to the EP is the data captured from the SUT execution by the test monitor. In the third and final step, the oracle procedure gives a verdict on test results by comparing the EP to the expected correct behaviour as expressed by the oracle information.

The following subsections give an overview of existing automation techniques to assist in test oracle creation, including both fully automated test oracles (as provided oracle components for a given domain) and automation tools to assist a user in oracle creation.

B. Automatic Test Oracle Components

In the context of automated test oracles, Daniel et al. [2] have presented a set of automated test oracles for refactoring engines. These oracles are based on the properties of the refactoring operations supported, including checking for the invertibility of the refactoring operation (performing the operation backwards to check it produces the original result), and checking that the moving of an element actually results in creating the item in a new location. This type of a test oracle is applicable to different refactoring engines, and Daniel et al. describe applying it on the Netbeans and Eclipse IDE's.

A generic approach for a test oracle is checking for thrown exceptions and application crashes [7]. A more domain specific but similar approach is presented by Mesbah and Deursen [3], who use invariants to define a set of automated test oracles for AJAX-based web-applications. They provide a set of invariant-based test oracles for generic properties of this type of web-applications, such as the HTML always being valid, and the DOM-tree not containing any (HTTP) error messages. These test oracles are then applicable to any AJAX web-application. As an oracle procedure they use an automated input-generation designed to crawl through web-pages and check that the resulting documents do not violate these invariants.

Memon and Xie [8] present an automated test oracle for GUI testing. This test oracle follows the traditional record/replay approach, where the properties of the GUI elements are used to describe its states. A model of the SUT behaviour is captured using a set of existing test cases that are assumed to describe the correct behaviour of the SUT, and a GUI state extraction technique. This model is then used as a basis for regression testing to describe the expected states.

Machine learning has been applied in several studies to generate a test oracle based on the EP of the SUT. These oracles are typically based on low-level EP data, such as capturing all function calls inside a program, their parameter values and the relations of these values [9]. A learning algorithm is trained with EP's labeled as failing and correct, which provides a test oracle that can classify a new execution as passing or failing. These oracles can be more generic than the previous approaches, but typically they need to be trained separately for each SUT and cannot test for any application specific behaviour, such as correct input-output transitions.

These examples summarize the type of support that current automated test oracle components can provide. In case

of Daniel et al. [2] the test oracles can be applied to different refactoring engines, but not to any other type of SW. More generic approaches are based on generic errors or exceptions thrown by the programming language constructs [7][9] or available in domain specific representations [3]. The Memon and Xie [8] approach is mainly applicable to regression testing only, with the assumption that the recorded model is correct, and the model can be fragile with regards to small changes in the SUT behaviour that are irrelevant from the test oracle perspective.

C. User Assisted Test Oracle Automation

A second category in automated test oracle creation support is in user assisted test oracle creation. Typically in these cases, the oracle procedure is provided and the user has to provide the oracle information. Usually, a basis for describing the oracle information is also provided in the form of tools or libraries that can be used to create or describe it.

In addition to providing automated oracle components (with both procedure and information) for the generic properties of web-applications as described earlier, Mesbah and van Deursen [3] also give the user the option to provide custom invariants to be checked, such as the contents of a table being update when a link is clicked. Their toolset will then automatically crawl through the web-application and check that the provided invariants are not violated. Here the invariants are the oracle information provided by the user and the checking of the invariants is the automated oracle procedure that is given. The toolset also provides means to describe the invariants, and in this way supports the creation of the oracle information.

Andrews and Zhang [10] have presented a technique for test oracle generation based on log file analysis. This is based the SUT writing a log file based using a predefined logging policy and a log file analyser asserting the correctness of the execution based on the log file. Their approach requires writing the log file analyser component and providing a matching logging policy to support the analyser. They illustrate the approach with a state-machine based matching, where the transitions are based on the available log lines. The log file analyser component is applied against log files collected from SUT execution, and makes the assertion whether the log file matches the expected behaviour or not. In this case the oracle information is provided in form of the analyser component that the user has to write. This is supported by the logging policy and the interfaces to their test execution system (the oracle procedure).

Both Ducasse et al. [11] and De Roover et al. [12] have described similar techniques for building test cases based on traces collected from a programs execution. They start with executing the SUT and collecting traces from the execution. Logic languages derived from Prolog are used to query the execution traces, and these queries act as the test oracles. The queries assert that the recorded behaviour matches the expected behaviour. Ducasse et al. [11] use the queries to filter relevant data from large, low-level, data sets, while De Roover et al. [12] do similar queries but aim at limiting the trace data to higher level events and lighter trace implementation. The aim with these techniques is to produce a model that is both human understandable and machine verifiable, in order to support both test automation and PC. In this case, the user has to provide the oracle information in the form of a query that describes what should be found in the (EP) trace. The oracle procedure is the test automation system that executes the queries and reports their results, doing a comparison against set expectations.

Lienhard et al. [13] describe the use of a visualization technique, based on a program execution trace, as a basis to assist the user in creating unit tests. This visualization is called the Test Blueprint. They focus their analysis on a part of a program execution, which they term an execution unit, in order to reveal so called side-effects. These side-effects describe the created and changed objects, changed object references and similar properties during the execution of the chosen unit. These are then provided to the user through their visualization, which helps the user in turning them into assertions. These assertions are used to verify that no important properties (side-effects) are violated during changes of the SUT. They also describe the visualization as supporting the creation of a test harness, as it shows required interactions with other objects. Creating the assertions in this case is done manually.

Program invariants are used as a basis for assisted oracle-generation in Agitator [14], Eclat [15] and in the technique proposed by Xie and Notkin [16]. These techniques require a set of program executions as a basis (such as existing test cases or an example program) and based on this create an invariant model to describe the SUT. This model is based on capturing all method calls and their parameter values (the execution profile). They then provide the user with the option of turning these invariants into assertions as part of the SUT unit test suite, to check that the invariants are not violated in regression testing. In this case, the user is actually provided with a form of a test oracle procedure and information. The oracle procedure is the assertion facility of the used unit test tool, and the oracle information is the invariants that are suggested to be turned into assertions. The procedure is not fully automatic, it requires the user to evaluate the usefulness of the proposed invariants, augment or modify them where needed and to choose which ones should be turned into assertions. As such, they can be provide highly automated support but, due to focusing on low-level execution data, are limited in their usefulness (much like the machine learning approaches described earlier) to unit- tests of small granularity classes or components. Higher level concepts, such as the properties of input-output transitions in relation to specification are not supported as such oracle information is in practice not embedded in program structure for invariant detection.

III. Program Comprehension

This section provides an overview of program comprehension concepts and related research to provide a basis for analysing the synergies with test oracles in later sections.

A. General Concepts

Program comprehension is a field dealing with human understanding of software systems, and its theoretical foun-

dations are based on fields studying human learning and understanding. The theories of PC are also referred to as cognitive theories of program comprehension [5], which highlights the purpose of PC techniques and tools as an aid to building human understanding of software systems. Program comprehension can make use of information from different sources, such as static analysis of program artefacts (e.g. source code) and dynamic analysis of program executions. In the context of this paper, when PC is discussed, it refers to techniques related to dynamic analysis.

Figure 2 shows the process of PC as a three-step process based on [17], adapted to include the impact of analysis of the program itself on the hypothesis of the program purpose. First, based on the program documentation, a hypothesis is made for what is the purpose of the program and how it is expected to work. This step can also be influenced by analysis of the program itself when the documentation is not up-to-date. The program is examined in the second step to build a hypothesis on how it operates. The building of this hypothesis can be influenced by the hypothesis on the program purpose. Finally, it is attempted to match the two hypotheses together to see if the understanding is correct. If this step fails, it forces a return to either step one or step two.

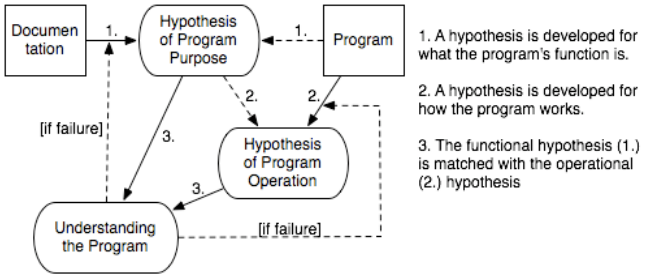

Figure 2. Program comprehension process.

In order to provide the required basis for how the application of PC techniques for test automation is demonstrated in the case study section later, the basic approaches for PC need to be reviewed. Two basic approaches to program comprehension are typically identified: top-down comprehension and bottom-up comprehension [5]. Both can be mapped to the process described in Figure 2, where the top down emphasizes documentation for step 1 and bottom-up emphasizes the program part. Further, from these a hybrid model called an integrated metamodel has been presented [18]. Other types of models include a knowledge based model, opportunistic and systematic strategies, and consideration of program and programmer characteristics [5].

The bottom-up approaches assume that the comprehension process starts from low-level concepts, such as reading source code statements, and grouping these into higher-level concepts [19]. In these models the programmer starts from composing small chunks to progressively larger chunks, finally acquiring a model for the program or its parts under investigation.

The top-down approach was presented by Brooks [20], who describes program comprehension as building knowledge about the problem domain and mapping it to the program source code. First an initial hypothesis is formulated based on the programmer's knowledge about the program domain. Based on information extracted from the program, the hypothesis is refined and subsidiary hypothesis can be generated. The verification of these hypotheses is based on beacons, which are described as sets of features (details) in the code that typically indicate the occurrence of certain structures or operations related to the hypothesis. It is seen that the investigation of a program will identify strong beacons for all hypotheses, and these beacons will lead to further refinement of the hypotheses.

In a hybrid approach, the programmer is seen to switch between these top-down and bottom-up approaches as seen necessary and as the analysis of the program progresses [18]. One is seen to move from the specification to source code and use all these available information sources as needed, and as described in the top-down and bottom-up approaches.

B. Existing Techniques

This subsection gives a brief overview of existing techniques related to PC with dynamic analysis. Since the intent is to provide a basis for mapping from PC to test oracle automation, the focus is on behavioral models as this allows matching them against test oracle requirements. As PC is a field with a large number of techniques and studies [21], the focus is only to give an overview of this area.

In order to support the human cognitive process of program comprehension, various tools and techniques have been presented with different approaches, and aiming different properties of the SW, such as structure and behaviour. These approaches include visualizations, pattern detection, summarization (e.g. clustering), data queries, and filtering or slicing the data [21].

Sequence diagrams are a popular means to model the behaviour of the analysed system [22][23]. These tools are intended to support functions such as mapping sequences of messages (method invocations) to features of the analysed SW (top-down approach), and to understand patterns of execution (bottom-up approach) [22].

State machines are another popular means of modeling SW behaviour, and many approaches to synthesize state-machines based on execution traces have been presented (e.g. [24], [25]). Although many of these list the support for PC as one of the uses, there are only few studies on actual use and how the generated state-machines assist humans in PC [21]. Although it is intuitive that understanding the states of a system and how the transitions between them happen help in PC, it would be useful to see empirical studies on how people actually use them to support this process.

Other types of models include invariant models [26], concurrency models [27], architectural models of components and connectors [28], and petri-nets [29]. These are described to help in tasks such as understanding the concur-

rency related dependencies [27][29] or structures of the system [28]. However, these too suffer from lack of empirical studies on how they would be systematically applied by their users. This lack of empirical studies (controlled experiments) on how the human users make use of these models is also highlighted as one of the lacking areas in PC research by Cornelissen et al. [21].

IV. Test Oracles and Program Comprehension

This section presents an analysis of the relations between the test oracle and program comprehension concepts presented in the previous two sections, and related research.

A. General Concepts

The previous two sections presented the basic concepts of test oracles and program comprehension and an overview of research done in these areas. This section reviews these two concepts together from the viewpoints of commonalities to use as a basis for finding synergies between them. Figure 1 and Figure 2 showed an overview of test oracle and PC procedures accordingly. Figure 3 shows these two figures at a higher abstraction level and maps them together. The top row shows the test oracle process and the bottom row shows the PC process.

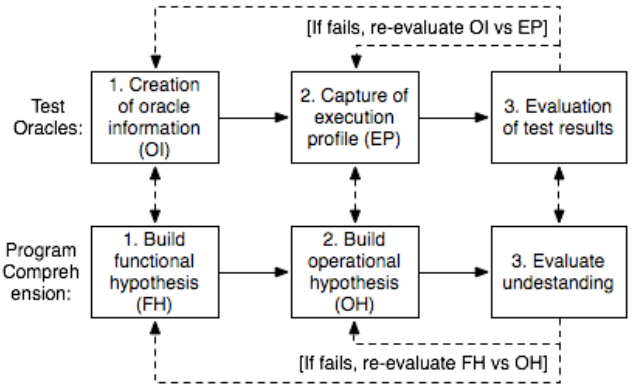

Figure 3. Test oracles and program comprehension.

Figure 3 shows how the two concepts are similar by mapping each of the three steps in both processes to each other and also providing a similar feedback loop in both processes. The details of these steps were described in the previous sections and are not repeated here. Instead this section is focused on discussing the conceptual similarities of these two processes.

In the first step, the creation of oracle information (OI) uses program documentation and execution profiles as input to create a specification of what is expected from the SUT. Similarly, in the first step of the PC process a functional hypothesis (FH) is built for the program based on its documentation and artefacts such as the EP. Thus, both processes have conceptually similar inputs, outputs and goals in this step.

In the second step of the oracle process, the EP of the SUT is captured. This EP describes the behavior of the SUT implementation. In step 2 of the PC process, the operational hypothesis (OH) is built to describe how the program operates. Again, both these processes have similar goals, inputs and outputs in this step. Both aim at building a model to describe what the program is and what it does. Both also use the program and its executions as input.

In the third step, the oracle process compares the OI model against the EP model to evaluate the test results. If these are found to match, the test result is marked as passed; otherwise it is marked as a failure. Similarly, in the program comprehension process, this step involves evaluating the FH against the OH. If these are found to match, the program comprehension is seen to be successful; otherwise it is seen to have failed.

Finally, in both processes, a failure in the third step prompts a return to the earlier model generation phases. As the evaluation is in both cases based on comparison of two models (or hypothesis), this step leads to the re-evaluation of both of these models to see which one(s) are not correct, refining these models based on this evaluation and repeating the process.

B. Existing Techniques

Many techniques that are mainly aiming to model a SW system list a number of possible fields where the authors think models generated from execution traces could be used. These fields usually include both PC and software testing. However, more concrete evaluations for all the included uses are in many cases missing, as usually a paper can only have one effective focus area. Despite this lack of studies, it is true that generated models at a higher level of abstraction than pure execution trace represented by function calls and parameter values are of course easier to understand for humans. As they also describe the executions of the SUT, they can be considered to have possible uses for software testing.

Some of the better examples are found in the research described in the earlier section about user assisted test oracles. For example, both Ducasse et al. [11] and De Roover et al. [12] describe their techniques as supporting both software testing and PC. They describe their tools from the software testing viewpoint, in the form that enables the user to create queries over the SUT traces and once satisfied with the answers, to turn these into assertion for the test suite of the SUT. PC is seen to be supported in answering the queries the user has about the program, and test automation in keeping these queries as a part of the regression test suite. As a part of the regression test suite, they can also be seen as upholding that understanding by reporting when the related assumption no longer holds.

Similarly, the previously describe work by Lienhard et al. [13] on their Test Blueprints technique aims to support the creation of test oracles with the aid of program comprehension techniques. The visualization they use is originally developed to support program comprehension, and in this case they also use it to help the user understand the SUT in order to create test assertions, which act as test oracles.

Invariant detection started out from work on producing models based on execution profiles and was described as potentially supporting many different domains, such as test automation and program comprehension [26]. A number of tools including Agitar [14], Eclat [15] and the technique

presented by Xie and Notkin [16] use these invariant based models as a basis and are described as supporting both software testing and PC. They exercise the SUT with a set of scenarios, either with generated input or with existing test cases. The inferred invariants of the SUT are then presented to the user and the user is given the option to turn these invariants into assertions to create new test cases along with related input data. PC is supported by describing the SUT behaviour as a set of invariants, test automation in allowing the user to turn these invariants into assertions for the test suite. Again, PC can also be seen to be supported by the inclusion in a regression test suite, which results in reporting when these assumptions no longer hold.

V. A CASE EXAMPLE

This section presents a case example of applying the concepts presented in this paper. Previously the theoretical background for the use of PC for user-assisted test oracle creation has been presented, including a brief description of a set of existing tools that can be seen to help in this process. This section is intended to present an example case of applying the previously presented theoretical concepts in practice and is thus termed a case example. With the help of PC tools and techniques, a model, including a test oracle, is generated based on the execution profile (traces) of the SUT. The concepts presented on the previous sections on the background theory are mapped to this concrete example, and it is shown how the PC concepts can help in the different parts of this process.

The SUT in this case is a component that acts as a database and a server for sensor data. Clients can connect to it, subscribe and query for sensor data and the server component keeps track of all the information received from different sensors. In this case, the captured execution profile of the SUT (the server component) is turned into a model for the model-based testing (MBT) tool ModelJUnit[1]. This means the SUT has to be modeled as an extended finite state-machine (EFSM), where it makes transitions from one state to another based on a set of guard constraints that describe when each transition can be taken. The model generation process has been implemented in a tool that automatically generates the model from the execution profile traces.

To start with, when a model is used as a basis for MBT of a SUT, the model should provide a comprehensive description of the SUT at the chosen abstraction level to ensure also comprehensive test generation from this model. As here the model is generated based on the execution profile captured from the existing SUT with a set of defined executions, the completeness of the model depends on the completeness of the execution profile.

As described by Cornelissen et al. [21], PC with dynamic analysis has two typical possible data sources to be used as a basis to analyse a system. One is the existing test suite and the other is any existing sample execution of the program, such as user sessions or example applications. In this case example, first the execution profile of the SUT has been captured by using an example application intended to demonstrate the use of the component. This application feeds about 20000 messages captured from real-world sensors through the component and sets up a set of test clients to illustrate the use of the data.

To check that the execution profile contains a comprehensive description of the SUT to be used as a basis for an EFSM, the execution profile is visualized with the ProM[2] process mining tool. This tool is intended to help people build an understanding of business processes by visualizing event logs as different types of models. By using it to visualize the execution profile trace as an FSM, the completeness of the execution profile can be checked against its specification (in this case written in technical English). This visualization is shown in Figure 4.

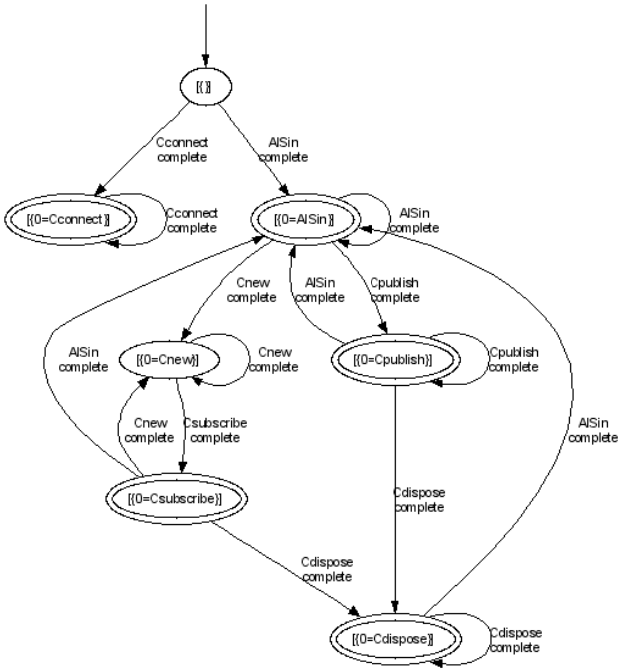

Figure 4. ProM visualization of the initial execution profile.

This visualization and analysis approach can be seen as a bottom-up approach to PC, providing means to gain an understanding of the available SUT execution profile. If only this view is used, it is not possible to tell if the execution profile is a comprehensive representation of the program behaviour for the purposes of testing. Instead, a hybrid strategy needs to be applied and the produced model compared against the SUT specification. From this it was seen that the execution profile was missing some important states and transitions, such as the ability to request sensor data and get a data message back as a reply. To address these issues, the execution profile was augmented with additional focused test cases (executions) intended to capture the missing behavioral states and transitions of the SUT. The resulting model for the execution profile with the example application and six focused tests is shown in Figure 5. This was judged to provide a comprehensive description of the SUT behaviour.

[1] http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

[2] http://www.processmining.org

With a satisfactory execution profile (describing required states) now available, it is used as a basis for generating the model in a format usable by the MBT tool. This shows another benefit of applying the synergies between PC and test automation as described in previous sections. At this point the tool support to build the basic state-machine for the test model is already available in the PC tool and can be reused for test model generation also. ProM is originally built for visualization to support analysis of event logs. However, in recent versions support for using the analysis algorithms from outside its GUI have also been added. Using this support, the test model generation tool implemented by the author of this paper creates the state-machine shown in Figure 5 and uses it as a basis in its own algorithms to generate the test model.

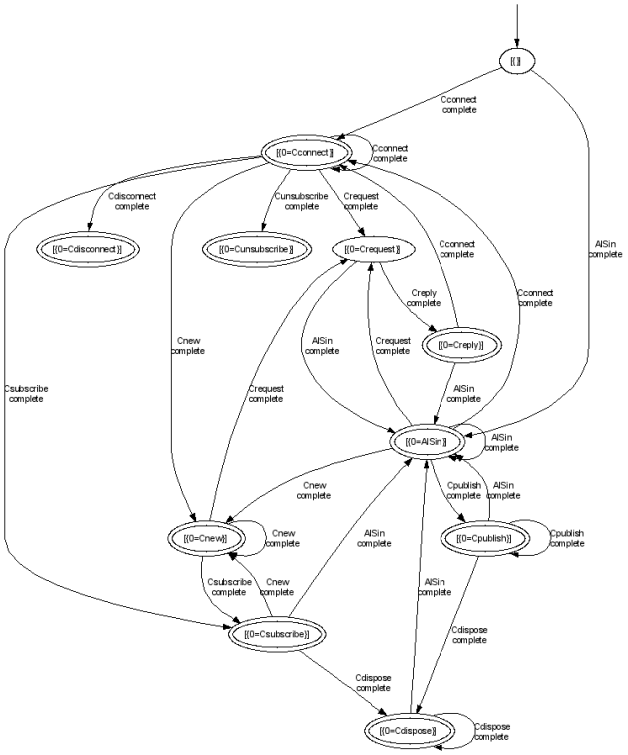

Figure 5. ProM visualization of the final execution profile.

The generated EFSM model includes everything needed by the MBT tool, including the test oracles. These oracles are generated from the ProM state-machine model and from the invariant model produced by the Daikon[3] tool, both generated from the EP trace. The generated test oracles check that the interactions in the tests generated by the MBT tool match those in the state-machine and use the invariants check for the correctness of return values. These need to be checked by the user manually to see that they do not miss any behavioral details, such as self-loops, or creation of complex objects.

Without going into details (due to space limitations of the paper and to retain focus), a basic approach to refine the generated model is to enable the SUT states and transitions one at a time and check their correctness against the specification. With this approach the building of the model also supports the PC process, as the MBT model can be executed and it exactly tells how it matches or differs from the actual implementation. This process is illustrated in Figure 6, Figure 7, and Figure 8. In Figure 6 the first state has been enabled. In Figure 7 two more states have been enabled. In Figure 8 one additional state has been enabled. These visualizations are provided by the MBT tool, which allows a feedback loop similar to using a PC tool. As the model refinement process progresses, all states will be enabled one at a time. At the same time, the executed model and the test oracles also give feedback back to the PC process. An abstracted model may hide important details, and when the MBT tool executes it and reports any errors in matching the model against the implementation, it also makes any assumptions in the model clear and checks them, reporting the results.

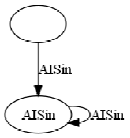

Figure 6. First state enabled in the MBT model.

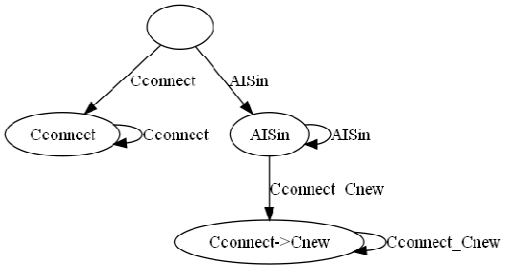

Figure 7. Three states enabled in the MBT model.

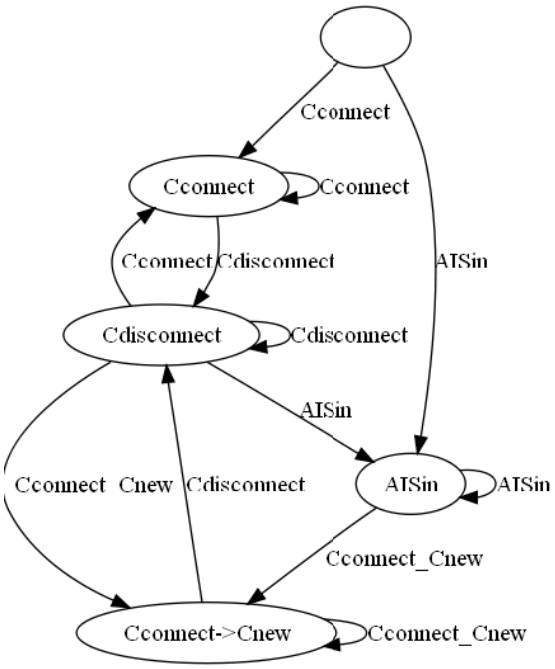

Figure 8. Four states enabled in the MBT model.

[3] http://groups.csail.mit.edu/pag/daikon/

During this case study, the usefulness of this approach was also demonstrated as it revealed six different, previously unknown, errors in the SUT. These were related to different properties of the SUT, such as incorrect implementation of the specification, missing or ambiguous parts of the specifications, and design errors in implementation details of the SUT.

VI. A Framework and Related Discussion

The existing techniques supporting both program comprehension and test oracle automation presented in section IV and the case example shown in section V share a number of properties. All start with supporting PC in the form of building a model based on the execution profile of the SUT. Test automation is supported by providing means to turn these models into different forms of test oracles for the SUT. From the PC viewpoint, they all apply a hybrid approach that starts with a bottom-up approach of building models from the execution profile, presenting them to the human user for analysis. The human user applies a top-down analysis by using the specification to determine the correctness of the proposed models and to turn them into test oracles.

A. A Framework for User-Assisted Test Oracle Generation

The different representations of the test oracle information used by the different techniques that have been presented also share the property of being invariant representations. Some of these techniques describe the provided oracle information as invariants [14][15][16], however they only discuss invariants as a basis provided by an external tool and not as the underlying concept of the test oracles themselves. However, in practice all oracle information is always a form of an invariant representation of some property of the SUT, which is then verified with the given oracle procedure. In the case of using queries over execution profile traces [11][12], the invariant is that the property expressed by the query holds in all analysed versions of the traces. In the case of the Test Blueprint approach [13], the invariant is that the "side-effects" checked by the created assertions are not changed. The concept of thinking of a test oracle information as a representation of an invariant, and the oracle procedure as an invariant-checker is important as it provides a conceptual framework for creating means to provide fully automated or user-assisted test oracle generation techniques.

In the presented approaches based on dynamic invariants inferred from the SUT execution profile, both the oracle procedure and the oracle information are provided, but the user needs to analyse the provided information and make an assessment if this is correct or not, possibly refining it [14][15][16]. Using queries of the execution profile trace as oracles requires one to define the queries as the oracle information to complete the oracle [11][12]. In the Test Blueprints approach, the user is presented with a visualization of the "side-effects" the execution of a program unit has, which can be used as a basis to write test oracles to verify these "side-effects" [13]. In the MBT model generation approach, both the oracle procedure and oracle information are provided and the user must check the information and refine the generated model as needed. All these approaches require the user to provide the oracle information or to refine it, while the oracle procedure is provided. The invariant- and model-based techniques can be seen as more advanced in their support for the user as they provide (generate) the initial (oracle information) model in a form directly executable as a test oracle, and which can then be analysed and refined by the user.

From these different techniques, it is possible to derive a set of guidelines for what to provide to the user when providing PC related techniques to assist in automated generation of test oracles. The items provide a framework for using program comprehension techniques to provide automated assistance for a user in generating test oracles, and can be summarized as providing the user with:

- An invariant notation suitable for the chosen oracle information.
- (a basis for) The oracle information, i.e. a set of invariants describing a meaningful properties of the SUT.
- The oracle procedure, i.e. an invariant checker.
- (an automated) Means to turn the oracle information into a test case with the oracle procedure.
- Assistance for the user to analyse (comprehend) the generated oracle information.
- Possibility to refine the oracle information.
- Means to (execute the model and) verify the complete oracle, i.e. an automated invariant-checker.

B. Related Discussion

The oracle procedure relates to the oracle information and how it needs to be processed and analysed. In many cases this can be simple, for example verifying that the output from a method call always has a value smaller than 100 (for invariant $x < 100$) can be implemented with a single assert statement. This is closely tied to the test automation platform used, such as a unit testing tool that provides the assertion facility, and the (program comprehension) tools used to provide the model of the execution profile that is used as a basis for the oracle information.

The basis for the oracle information as described in the techniques reviewed in this paper is formed from the invariant- and state-machine models of the SUT created based on the execution profile. Thus they already provide an abstraction generated based on the execution profile that the user can turn into or refine to produce the required oracle information.

Turning the oracle procedure and information into a test case requires mapping the oracle information to the oracle procedure. For example, in the MBT case example, this requires parsing both the invariant and state-machine models and turning them into an EFSM model. In the context of using PC tool, when they provide access to their internal model representations this provides the best support for both PC and test oracle automation as shown in section V.

As any tests generated in this way are based on the execution profile and are thus limited by what executions it contains, the user must be able to analyse the provided test

oracles and to refine the model to match the specification. Since the model describes the actual behaviour of the SUT, the user must also be able to verify that this is actually the expected behaviour of the SUT as expressed by its specification. This highlights the need for means to analyse the models with tools such as those used in PC. This was demonstrated in the MBT case example, where the resulting state-machine could be visualized both from the execution profile and after being generated into an EFSM. In the same case example, it was demonstrated how the model assumptions can be made clear and verified (comparing the generated test oracle vs. the specification) by the PC related visualization and analysis, and by executing the model with the test automation tool.

The process as described by the guidelines matches that of both program comprehension and test oracle automation as described in Figure 3. The user starts with the model of what is the expected behaviour of the SUT, such as its specification. Using the tools provided, the execution profile is captured and turned into a model to be used as a basis for defining the oracle information. The user analyses this information, refines the model and verifies its correctness. As required, both of these models are iteratively refined according to findings in the verification phase.

These examples show that using PC related concepts and techniques can be helpful in the context of providing automated support for test oracle generation. However, as discussed before, and also noted by Cornelissen et al. [21], there is a lack of studies on how the PC techniques support actual humans in their work. This also makes it more difficult to take these approaches and consider how they could be applied in the context of test oracle generation. The example shown in section V uses PC techniques to generate and visualize state-machines as a basis for generating a test model, including a test oracle. Although there are tools for state-machine generation in PC, there are not many empirical studies on their use [21]. Still, in the case study described in this paper, one tool was used to aid in test oracle generation with a straight-forward approach as described in section V. However, more comprehensive studies in application of PC techniques would make the process of applying them for test oracle automation easier. Similarly, this could be eased by providing support for accessing and using the models provided by the PC tools externally from other tools. This support and the framework presented in this section also provide a basis for creating more powerful techniques for user-assisted test oracle generation.

VII. Conclusions and Future Work

In the context of test automation, the creation of test oracles is one of the most difficult parts to automate. This is also visible in the limited number of papers that address the test oracle automation problem. This paper addressed this issue by providing a framework for applying techniques from the field of program comprehension to provide automated assistance to the user in creating test oracles. Related work on this topic was reviewed, analysed, and brought together with the concept of program comprehension for test oracle automation. The concept was illustrated with a practical example of generating models usable as a basis for test oracles in model-based testing. By comparison of this example with related work, a framework was provided for applying program comprehension techniques to provide automated assistance for users in creating test oracles. The provided framework describes test oracles as invariant-checkers and provides a set of guidelines for providing automated assistance to the user in generating these invariant-checkers based on information captured from the program execution with the help of dynamic analysis techniques. The provided framework helps with providing more powerful techniques to assist in the test oracle generation process.

This paper also highlighted need for future work in the field of program comprehension to identify how a human user actually makes use of a program comprehension technique. This information is needed in order to automate the use of these techniques as much as possible in the context of the framework presented here. In the field of test automation, interesting future work includes making use of more program comprehension techniques to support user-assisted test oracle generation.

Acknowledgment

This work has been supported by the Nokia Foundation. The author wishes to thank Eric Verbeek for his help with the ProM tool, and Andy Zaidman and the anonymous reviewers for their helpful comments on improving the paper.

References

[1] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Proc. Future of Software Engineering (FOSE'07)*, 2007.

[2] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'07)*, Dubrovnic, Croatia, 2007, pp. 185-194.

[3] Ali Mesbah and Arie van Deursen, "Invariant-Based Testing of Ajax User Interfaces," in *Proc. 31st Int'l. Conf. on Software Eng.*, Vancouver, Canada, 2009.

[4] H. Sneed, "Program Comprehension for the Purpose of Testing," in *Proc. 14th Int'l. Workshop on Program Comprehension (IWPC'04)*, 2004.

[5] M-A. Storey, "Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future," *Software Quality Journal*, vol. 14, no. 3, pp. 183-208, Sept. 2006.

[6] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley, "Specification-Based Test Oracles for Reactive Systems," in *Proc. 14th Int'l. Conf. on Software Eng. (ICSE'92)*, Melbourne, Australia, 1992, pp. 105-118.

[7] X. Yaun and A. M. Memon, "Using GUI Run-Time

State as Feedback to Generate Test Cases," in *Proc. 29th Int'l. Conf. on Software Eng. (ICSE'07)*, 2007.

[8] Atif Memon and Qing Xie, "Using Transient/Persistent Errors to Develop Automated Test Oracles for Event-Driven Software," in *Proc. 19th Int'l. Conf. on Automated Software Eng. (ASE'04)*, 2004.

[9] Murani Haran, Alan Karr, Michael Last, Alessandro Orso, Adam A. Porter, Ashish Sanil, and Sandro Fouché, "Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks," *IEEE Transactions on Software Eng.*, vol. 33, no. 5, pp. 287-304, May 2007.

[10] J. H. Andrews and Y. Zhang, "General Test Result Checking with Log File Analysis," *IEEE Transaction on Software Eng.*, vol. 29, no. 7, pp. 634-648, July 2003.

[11] S. Ducasse, T. Gîrba, and R. Wuyts, "Object-Oriented Legacy System Trace-Based Logic Testing," in *Proc. European Conf. on Software Maintenance and Reengineering (CSMR'06)*, 2006.

[12] C. D. Roover, I. Michiels, K. Gybels, K. Gybels, and T. D'Hondt, "An Approach to High-Level Behavioral Program Documentation Allowing Lightweight Verification," in *Proc. 14th Int'l. Conf. on Program Comprehension (ICPC'06)*, 2006.

[13] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz, "Test Blueprints - Exposing Side Effects in Execution Traces to Support Writing Unit Tests," in *Proc. 12th European Conf. on Software Maintenance and Reengineering (CSMR'08)*, 2008, pp. 83-92.

[14] M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," in *Proc. Int'l. Symposium on Software Testing and Analysis (ISSTA'06)*, Portland, Maine, 2006, pp. 169-179.

[15] C. Pacheso and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *Proc. European Conf. on Object-Oriented Programming (ECOOP'05)*, 2005, pp. 504-527.

[16] T. Xie and D. Notkin, "Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions," *Journal of Automated Software Engineering*, vol. 13, no. 3, pp. 345-371, July 2006.

[17] K. Magel, "A Theory of Small Program Complexity," *ACM SIGPLAN Notices*, vol. 17, no. 3, 1982.

[18] A. von Myrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer*, vol. 28, no. 8, pp. 44-55, August 1995.

[19] N. Pennington, "Stimulus Structures and Mental Represntations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.

[20] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *Int'l. Jounral of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[21] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A Systematic Survey of Program Comprehension through Dynamic Analysis," *IEEE Transaction on Software Eng.*, 2009.

[22] C. Bennett, D. Myers, M-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A Survey and Evaluation of Tool Features for Understanding Reverse-Engineered Sequence Diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 291-315, July 2008.

[23] L. C. Briand, Y. Labiche, and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Transactions on Software Eng.*, vol. 32, no. 9, pp. 642-663, Sept. 2006.

[24] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Proc. 30th Int'l. Conf. on Software Eng. (ICSE'08)*, Leipzig, Germany, 2008, pp. 501-510.

[25] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe, "Automated Discovery of State Transitions and their Functions in Source Code," *Software Testing, Verification and Reliability*, vol. 18, no. 2, pp. 99-121, June 2008.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Eng.*, vol. 27, no. 2, pp. 99-123, Feb. 2001.

[27] J. E. Cook and Z. Du, "Discovering Thread Interactions in a Concurrent System," *Journal of Systems and Software*, vol. 77, no. 3, pp. 285-297, Sept. 2005.

[28] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan, "Discovering Architectures from Running Systems," *IEEE Transactions on Software Eng.*, vol. 32, no. 7, pp. 454-466, July 2006.

[29] W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther, "Procee Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting," *Software and Systems Modeling (SoSyM)*, 2009.