Towards Anti-Model-based Testing

Antonia Bertolino and Andrea Polini
Istituto di Scienza e Tecnologie dell'Informazione
"A. Faedo" (ISTI-CNR)
Area della Ricerca CNR di Pisa 56100 Pisa, Italy
[antonia.bertolino, andrea.polini]@isti.cnr.it

Paola Inverardi and Henry Muccini
Dipartimento di Informatica
Universitá dell'Aquila
Via Vetoio 1, 67100 L'Aquila, Italy
[inverard, muccini]@di.univaq.it

Software testing refers to the dynamic verification of a system's behavior based on the observation of a selected set of controlled executions, or test cases [2]. While in traditional approaches test cases were selected aimed at covering the program source code [5], nowadays we can apply a variety of testing techniques all along the development process, by basing test selection on different pre-code artifacts, such as requirements, specifications and design models [2].

Model-based testing consists in deriving a suite of test cases from a model representing software behavior. Such a model can be generated from a formal specification or designed by software engineers through diagrammatic tools. In principle, the derivation of the test cases can be done automatically, and indeed several approaches have been recently proposed that do this starting from models in different languages, e.g., [9, 1, 6]. By executing the model-based test cases, the conformance of the implemented system to its specification can be validated.

Model-based testing is certainly useful and effective; however, there can be several reasons why such an approach *cannot be applied* or is *too expensive* for deployment in a specific context. One generic barrier to a wide adoption of model-based testing is its *inherent complexity*, which requires a deep expertise in formal methods, even where tool support is available – as testified by case studies in the AGEDIS project [1]. Another obstacle is the difficulty in *forcing the implementation to take a defined path as identified in the model derived test sequences*. The latter are generally expressed at an abstract level, while the executable test cases must be more concrete and more informative (e.g., [3]). Finally, one further counter-motivation to the practice of model-based testing can be the use of *legacy systems* or *COTS*, for which behavior models are not available.

Considering in particular *component-based software development*, a system is generally obtained by assembling already existing components, for which we cannot a-priori assume that a specification or the source code are available. In such cases, model-based testing is not applicable, or would be too costly. We assume in fact that the system assembler has a high-level specification of the global architecture, but can only pose in practice very basic requirements on the behavior of the acquired components.

This is the rationale for an *"anti-model-based testing approach"* as the one we outline in this paper. While model-based testing starts from an a-priori established model and tries to execute some sequences derived from this model, in "anti-model based" testing we take the opposite direction. We execute the implementation on some sample executions, and by observing the traces of execution we try to infer/synthesize a-posteriori an abstract model of the system. In both cases, model-based and anti-model based, we end up having a model on one side and an implementation on the other, and we want to decide on the basis of test executions whether the implementation is acceptable by its comparison with the model.

To reverse engineering a model from the test traces we need to apply two technologies: first, we have to *reverse engineering some scenarios* from the execution traces (as done for different reasons in other research work (e.g., [4]); second, from the so obtained scenarios, e.g. in form of UML sequence diagrams, we synthesize a behavior model.

Figure 1 graphically summarizes the approach we are working on:

Assumption: we assume to deal with a component-based system, i.e., an assembly of component-based, black-box components. A component specification is missing, as the source code itself;

Step 1: Derive the usage profile based on a high-level specification of the global architecture;

Step 2: Launch the test cases and monitor the traces;

Step 3: Reverse-engineering of a set of (meaningful) sequence diagrams, in order to synthesize a behavior model.

In detail, when a software system has to be produced through the assembly of components, wanted system requirements needs to be identified and specified. Whenever the main system requirements are elicited, we may start identifying the architectural components. We can thus buy

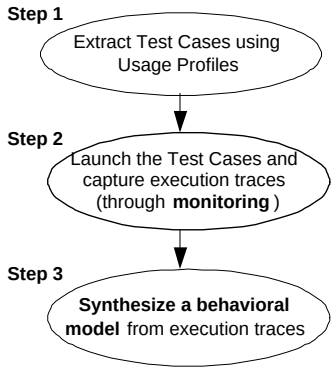

Figure 1: Approach activities

the components and create the glue code as a way to produce the desired system.

In Step 1, suitable test cases have to be identified. As the basis assumption of this approach is that a component model is not available, we use the only information that is anyhow available (it may be in various forms): the expected Input/Output functions of the components. This information has to be available in some form, otherwise we could not even use the components. In other words, as a very minimum the component user must know how to solicit the components and what to expect as a reaction. To make such an approach systematic, we will stimulate the component interactions by trying to reproduce the operational usage, i.e. we will try to reconstruct an usage profile.

In Step 2, we have to launch the test cases and monitor the execution traces. Goal of this step is to stimulate the system with inputs, capturing information on execution traces. The idea of capturing traces from code execution is not new. In particular, many strategies aimed to reverse-engineer dynamic models are reported in the literature, many of them surveyed and compared in [4]. The general idea is to instrument the source code, by adding some monitors, and run it with some inputs. The monitors help collecting relevant information on run-time execution, such as methods execution, classes and/or objects communication, control or data flow information.

What makes our monitoring activity different is the assumption that components are black-box and their specification is missing. This assumption strongly impacts the way monitoring can be performed. In our context, information is gained by instrumenting the glue code used to assemble the components. The information we need to collect regards the interaction between components requiring or providing services. Therefore the tracing mechanism should be able of recording each invocation made by one component to another component. This could be easily obtained through the use of specific wrappers used to trace the incoming and the outgoing calls for each component.

In Step 3, the execution traces collected in the previous step are used to synthesize a behavioral model. This is the most interesting aspect of this approach. In fact, in order to synthesize state machines from execution traces, our idea is to extract scenarios from the execution traces and eventually use such scenarios to synthesize state machines, reusing existing synthesis algorithms (e.g., [7, 8]). An execution trace, in fact, may be considered as the interleaving of different scenarios.

Concluding, model-driven approaches have been recently utilized by software engineers for analysis and testing purposes with promising results. Unfortunately, such approaches cannot be applied whenever the system model is unavailable, such as in the assembly of COTS. Goal of our research is to propose some alternative approaches in these cases; even when system models and software code are not available, an anti-model-based testing technique might produce useful results.

In this paper we simply illustrated how a reverse-engineered model can be produced by analyzing execution traces derived via operational testing. In future work we will also investigate how such a reverse engineering process may help discover unexpected behaviors. In particular, our next steps will be directed to evaluate, through model checking techniques, how much the implementation is good with respect to expected qualities. Moreover, we will analyze if the system model so obtained contains unexpected behaviors. If it does, we may gain some information on how good the selected test cases are.

References

[1] AGEDIS Project. http://www.agedis.de/index.shtml.

[2] A. Bertolino. Software Testing. In SWEBOK: Guide to the Software Engineering Body of Knowledge,IEEE.

[3] A. Bertolino, P. Inverardi, and H. Muccini. An Explorative Journey from Architectural Tests Definition downto Code Tets Execution. In *IEEE Proc. Int. Conf. on Software Engineering* (ICSE2001), pp. 211-220, May 2001.

[4] L. C. Briand, Y. Labiche, and Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. In 10th Working Conference on Reverse Engineering. November 2003, Victoria, B.C., Canada.

[5] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Trans. on Software Engineering, SE-11 (1985), pp. 367-375.

[6] J. Ryser, and M. Glinz. Using Dependency Charts to Improve Scenario-Based Testing. Proc. of TCS2000 Washington D.C., June 2000.

[7] UBET, http://cm.bell-labs.com/cm/cs/what/ubet/.

[8] S. Uchitel, J. Kramer and J. Magee. Synthesis of Behavorial Models from Scenarios. IEEE Transactions on Software Engineering, Vol. 29, Number 2, February 2003.

[9] UMLAUT Project.
Available at http://www.irisa.fr/UMLAUT/.