A generic knowledgebase for test generation

Artem Kotsynyak, Andrei Tatarnikov
Software Engineering Department
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
Moscow, Russian Federation
Email: {kotsynyak, andrewt}@ispras.ru

Abstract— **Nowadays a lot of various test generation tools are developed and applied to create tests for both software applications and hardware designs. Taking into account the size and complexity of modern projects, there is an urgent need for "smart" tools that would help maximize test coverage and keep the required effort and time to a minimum. Despite the fact that each project is unique in some sense, there is a set of common generation techniques that are applied in a wide range of projects (random tests, combinatorial tests, tests for corner cases, etc). In addition, projects belonging to specific domains tend to share similar test cases or use similar heuristics to generate them. A natural way to improve the quality of testing is to make the most of the experience gained working on different projects or performing testing at different stages of the same project. To achieve this goal, a knowledgebase holding information relevant to test generation would be of a great help. This would facilitate reuse of test cases and generation algorithms and would allow sharing knowledge of "interesting" situations that can occur in a system under test. The paper proposes a concept of a knowledgebase for test generation that can be used in a wide range of test generation tools. At ISPRAS, it is applied in test program generation tools that create test programs for microprocessors. The knowledgebase is designed to store information on widely used test generation techniques and test situations that can occur in a microprocessor design under verification.**

Keywords— test generation, testing knowledge, test reuse, test situations, constraints, knowledgebase.

I. INTRODUCTION

To start with, it should be said that our team works in the area of hardware verification [1, 2]. Therefore, the main focus of the research is on generating tests for hardware devices. However, the concepts described in this paper are not limited to hardware verification and remain relevant to a wide range of domains.

Testing accounts for up to 70% of overall project resources. To reduce the expenses, an effort is made to automate the testing process. Over the recent couple of decades, approaches to automated testing have evolved significantly. Still, increasing complexity of modern projects demands for more efficient methods. To get the big picture of the state of the art, let us first consider existing approaches from the most trivial to the most advanced.

The most straightforward way to automate test generation for one's project is to write a simple test generator in one of the popular programming languages. Such generators are usually targeted at producing random or combinatorial tests. However, they can also include heuristics that help generate tests for some "interesting" situations (e.g. boundary conditions). This approach has the following obvious disadvantages: such tools are inflexible and the knowledge they include is unsuitable for reuse as it is usually hardcoded. Moreover, random and combinatorial tests are not systematic and cannot guarantee a sufficient level of test coverage.

To cover nontrivial cases that are unreachable by using random and combinatorial generation, a test generation tool should be strengthened to be able to create directed tests [3]. Directed tests are usually generated on the basis of test templates that provide abstract high-level descriptions of testing problems. Such an approach is called template-based generation. Test templates use constraints to formulate conditions of occurrence for situations to be covered. Briefly speaking, constraints are a set of formulae describing relations between data (i.e. properties to be held for some events to fire). One of the advantages of template-based generation is that it separates test generation logic from the description of specific test cases, which simplifies test maintenance. More importantly, this allows constraints to be reused in other tests. However, the reuse is limited as constraints are described in terms of the verified system (hardware design or software application) and are not systematized. The issue is that manual creation of complex constraints is quite laborious and it might require a significant effort to adapt them for a different system. In fact, this could be improved as constraints for similar verification tasks tend to share common parts.

The next step in the evolution of approaches to test generation is model-based generation [1, 2, 3]. It implies separation of knowledge of the verified system's configuration from knowledge of test generation techniques. The former is referred to as a model and the latter is often called testing knowledge [5, 6]. The model can be either created manually or built automatically on the basis of formal specifications. The advantage of this approach is that it allows describing test cases in terms of the model, which results in more abstract descriptions. In addition, the model often includes coverage information that can be extracted from formal specifications or from other sources. In a nutshell, to generate high-quality tests, two types of knowledge are required: (1) knowledge of the verified system's configuration to be able to generate valid tests and (2) knowledge about situations that can occur in the system to be able to generate tests that would hit all "corners" of that system. Coverage information can be represented by a set of constraints describing conditions for various test situations. In this case, to generate test programs for the target system, one needs to provide a test template specified in terms

of information exposed by the model and constraints describing corresponding situations. As it can be noticed, to provide a good quality of test coverage, it may require creating a significant amount of test templates describing test cases for all possible situations. When this job is done manually, it can be time-consuming and there is a chance to miss some "interesting" cases especially when the constraints are not systematized and the coverage model changes as new knowledge about the system is acquired.

To further automate the process of test program generation, constraints need to be stored in a systematized way. In other words, knowledge of "interesting" situations and knowledge of how to obtain data causing these situations to fire should be accumulated in a knowledgebase for further use in the test generation process. Also, it would be highly desirable to have this information stored in a human-readable form to simplify its reuse and the maintenance of the knowledgebase. This leads to an idea of a knowledgebase that would store testing knowledge including commonly used constrains, algorithms for solving them, algorithms of random and combinatorial test data and test sequence generation, methods of exploring properties of the verified system's model, etc. Having this knowledge stored in a systematized way will allow making more intelligent decisions during test generation. One of the main goals is to reduce the number of test templates. The use of the knowledgebase would allow creating some of them in an automated way, therefore reducing the effort and increasing the coverage quality. Also, having a centralized store of testing knowledge gives a great advantage in terms of reuse and sharing experience between test engineers.

As the project the verification team is working on moves from the requirement elicitation to the release, more and more testing knowledge is accumulated. It may come from different sources such as requirements, specification, expertise, failed tests, automated analysis, etc. Some of this knowledge can be presented in an abstract way so that common test cases like overflows and other could be reused in projects with similar components. A centralized store helps ensure that each test engineer has this knowledge in hand and no "interesting" situation is ignored.

The present paper describes concepts of a knowledgebase for test generation. The knowledgebase is being developed at ISPRAS to be used in projects dedicated to hardware verification [1, 2].

The rest of the paper is organized as follows. Section II gives an overview of existing works related to testing knowledge. Section III provides a list of core requirements for the knowledgebase. Section IV describes the architecture of the knowledgebase and explains how it can be integrated with test generation tools. Finally, Section V concludes the paper.

II. Related work and motivation

Methods of efficient test generation have always been a major subject of research. One of the most important applications is functional verification of microprocessors where test program generation and simulation is the most common approach applied at the system level. Due to enormous complexity of modern microprocessors and severe time-to-market pressures, it is quite a challenging task. For this reason a lot of effort has been invested to maximize automation of this activity. This resulted in the emergence of a great number of test generation techniques. Also, a significant amount of knowledge about bug-prone areas in hardware designs has been accumulated. An important direction is to systematize the accumulated knowledge to further automate the test generation process and reduce its cost by facilitating knowledge reuse.

IBM Research [3, 5, 6] has been one of the main contributors in the field of test program generation for microprocessors during the last decades. The first test generation tools were developed in the middle of 1980s. Test program generators by IBM Research have evolved over time from random to directed model-based generation schemes. Genesys-Pro, one of the most recent tools, uses test templates that describe test generation problems as constraint satisfaction problems and uses a generic constraint solver customized for pseudorandom generation to increase the coverage quality. Constraints are based on the architectural description captured by the model and on the testing knowledge representing a set of methods that help increase the quality of generated test cases. There are two types of constraints: (1) mandatory ("hard") and (2) non-mandatory ("soft"). Constraints that originate from architectural description are typically marked as mandatory. "Soft" constraints help shift the bias of the generated stimulus to make test cases more "interesting" and can be ignored if the solver fails to find a solution. Testing knowledge, as it is described in papers by IBM Research, represents a collection of architecture-independent constraints and constraints specific to a given design. Also, it includes a set of heuristics that use accumulated knowledge of the semantics of the verified design to shift bias towards specific constraints to maximize coverage. IBM Research does not reveal details on how exactly the storage of testing knowledge is organized and integrated with their tools. However, their testing knowledge is obviously oriented only towards test program generation for microprocessors and is likely to be tightly coupled with their test generation tools. Two important aspects that were not covered in their papers are: (1) systematization of constraints and (2) means of combining constraints to describe complex problems (this particularly applies to constraints of different types). It is possible to specify probability distributions between "soft" constraints in a test template. However, there are reasons to think that no facilities are provided to do this at the level of testing knowledge.

Another company that has made a significant contribution in development of test generation tools is Obsidian Software (now acquired by ARM) [4]. The company specializes in development of verification and validation software used in the design of microprocessors. Their test program generation tool RAVEN (Random Architecture Verification Engine) is able to generate random and directed tests based on test templates. To achieve a better coverage, it makes use of coverage grids and accumulates verification knowledge in a database. Test templates are focused on the coverage grid and use constraints that allow RAVEN to intelligently choose random values to reach specific coverage goals. Unfortunately, documentation available on the tool does not provide detailed information on how the mechanism of knowledge accumulation is organized.

The motivation of the present research is to work out the concepts and to design the architecture of a knowledgebase for test generation that could be used in a wide range of test generation tools. It should help systematize various types of testing knowledge and facilitate its accumulation and reuse. The paper aims to contribute to the research in the field as the lack of information on competitors' solutions makes it difficult to apply their ideas. The paper summarizes the ideas from different sources [4, 5, 6], proposes some important improvements and expresses our vision for organization of a knowledgebase for test generation.

III. Requirements for the knowledgebase

The knowledgebase should maximize the quality of test coverage and minimize the effort required to create tests. For this purpose it accumulates knowledge about different test situations (conditions that make them fire, probabilities of their occurrence, methods of producing corresponding stimuli, etc.) This creates a possibility to easily create complex test cases by combining the accumulated knowledge. If this job is automated, it will help reduce the number of test cases described manually, therefore increasing the productivity of the verification team. Here is the list of the main requirements a generic knowledgebase for test generation should satisfy to achieve its goals:

1) The knowledgebase should be able to store and accumulate testing knowledge of a wide range of types coming from various sources and having different formats. This includes sets of test values, commonly used generation algorithms, constraints, methods of combining them, heuristics for shifting biases, etc.

2) The stored knowledge should be systematized and organized into a hierarchy. This will simplify its maintenance and reuse and will allow extracting common components.

3) It should be possible to easily integrate the knowledgebase into test generation environments of different kinds. The client environment should be provided with full access to the accumulated knowledge. To facilitate it, the knowledgebase should be implemented as an open-source project.

4) The knowledgebase should facilitate the transfer of project-independent knowledge between projects in a similar domain. This applies to test situations, constraints, data generators, etc.

IV. Architecture

The most important components of the knowledgebase architecture are the *storage engine*, *selector* and *resolution module* as shown in Figure 1. Further in this section, they will be discussed in more detail.

The job of the *storage engine* is to provide a persistent storage for any kind of knowledge allowed. The storage engine can be powered by any database technology. To add new knowledge or alter existing, users should interact with the engine via the *control interface* built on top of it. The interface provides access to logical representation of the stored knowledge hiding any details about the underlying database and data organization along with normalization logic specific to the knowledgebase implementation.

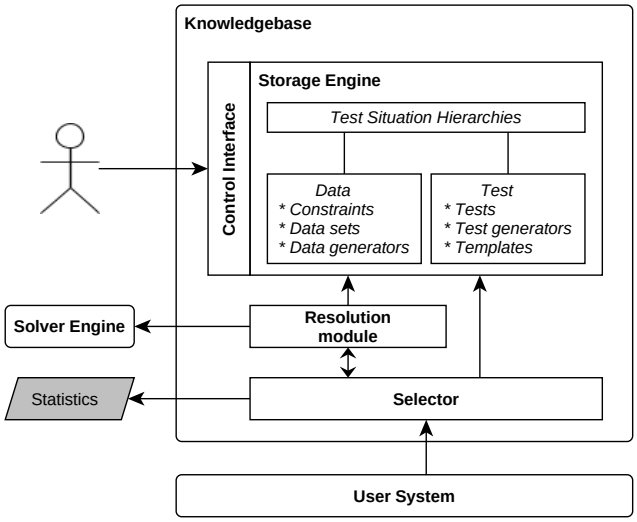

Figure 1. Architecture scheme

The main responsibility of the knowledgebase is to store hierarchies of test situations along with associated knowledge. Hierarchies of test situations being part of testing knowledge themselves are used as a tool for organizing the acquired knowledge. Therefore, it is up to the storage engine to store, manage and provide access to these structures to the rest of the knowledgebase modules. Nevertheless, the storage engine treats knowledge as data and does not implement any additional logic beyond what is encapsulated in the control interface.

Basically, a test situation in a hierarchy is a symbolic representation of an event (or a group of events) that can occur in the system under test, but the hierarchy itself does not provide any information about what kind of event it is. Hierarchies are represented by directed acyclic graphs (DAG) where nodes specify test situations while arcs denote refinement relation, i.e. if there is a path from node *u* to node *v* then node *v* represents a situation that is a special case of the situation of node *u*.

There are two types of situation hierarchies: *abstract* and *concrete*. Abstract hierarchies are used to describe commonalities between projects belonging to the same domain, while concrete hierarchies specify relations between particular test cases. Representations of abstract and concrete hierarchies have several important differences. First, abstract hierarchies are represented by unweighted DAGs, while weighted DAGs are used for concrete hierarchies where arc weights denote the desired probabilities of corresponding events. Second, nodes in a concrete hierarchy can be associated with additional knowledge about situations represented by these nodes (e.g. constraints describing the conditions for corresponding events to fire).

Figure 2 shows an example of a simple situation hierarchy that specifies situations from the microprocessor verification domain. The "add" node denotes situations possible in the

execution flow of an addition instruction and the “overflow” node denotes any kind of an overflow situation. Also, there is a refinement for an integer overflow called “int_overflow” and two explicit terminal situations called “corner” and “zero_sum” (the former describes a corner case for the integer overflow situation and the latter specifies the zero-sum situation). It is shown that the “int_overflow”, “corner” and “zero_sum” situations are associated with constraints describing data resulting in corresponding events. Implicit situations for the normal flow and random values are omitted along with the probabilities of their occurrence.

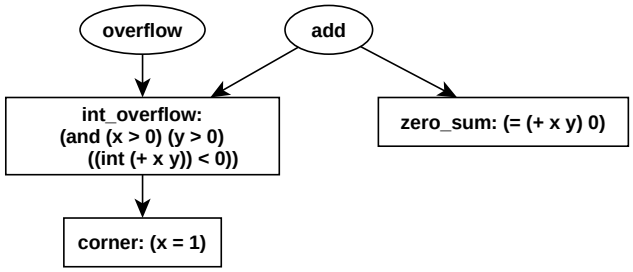

Figure 2. Example of a situation hierarchy

Arc weights in a concrete hierarchy describe relative rates at which specific test situations are to be obtained during test generation. Therefore, this can be used to control the generation process since it is allowed to bias probabilities in order to get behavior varying from fully random to fully deterministic. Moreover, zero probability effectively removes test situations from being exploited in test generation and can be used to disable unimplemented or irrelevant features of the current system under test.

Associated knowledge in concrete hierarchies may vary from complete tests to abstract templates. It can be stored in a database or in plain files. One of our goals is to reuse existing test generators so it is allowed to use them as associated knowledge via appropriate adaptors. This knowledge is primarily used in the test generation process and the querying system should be able to handle it by itself.

The control interface of the storage engine provides mostly database editor functionality. Therefore, to handle queries to the knowledgebase with respect to knowledge semantics, a specific module called *selector* has been introduced. The module “knows” about knowledge organization and uses probabilities stored within concrete hierarchies to select specific knowledge. Since every query to the knowledgebase is passed through the selector, it can trace test situations queried and produce a statistical report that can be used to adjust situation probabilities or to perform coverage analysis.

In the simplest cases, the selector just fetches the stored data and passes it to the querying system. This works for flat data and tests, but not for generators and constraints. To handle these correctly, an additional component called the *resolution module* has been included. Its initial purpose is to run generators stored as knowledge or to pass constraints to some external solver to produce data. The resolution module is designed to be an extension mechanism that has access to all internals of the knowledgebase and is allowed to run external applications. It is used whenever the selector decides that knowledge requires additional treatment before being sent to the querying system. Therefore, it can be adjusted in a domain-specific way to handle much more sophisticated scenarios, e.g. generate tests on the fly if a test template has been queried.

It should be noted that despite the fact that the knowledgebase considers test generators and tests as knowledge it is not a test generation system. Generating tests using the knowledgebase is straightforward and can be done at different levels depending on the contained knowledge and its organization, e.g. in microprocessor verification we can potentially generate test data for a single instruction, for a complex test template or generate a final test program using stored tests and generators. The test generation system queries the knowledgebase for test situations that correspond to terminal or non-terminal nodes in the hierarchy. In the latter case, the selector will use some refinement of the situation given with respect to probabilities stored within the hierarchy. Either the selector is able to fetch knowledge by itself, or it delegates the task to the resolution module, or both of them fail because the storage engine does not contain knowledge required or resolution marks the query unsatisfiable. In a successful scenario, the output is a test or test data and it is up to the querying system to distinguish between them.

V. CONCLUSION

We have proposed the concept and the architecture of a generic knowledgebase for test generation. The knowledgebase can be used in a wide range of test generation tools to accumulate knowledge related to the system under test. At ISPRAS, it will be integrated with tools responsible for test program generation for microprocessors. It facilitates knowledge reuse and allows making "smart" decisions during the process of test generation based on the accumulated knowledge. This helps improve test coverage and simplify test development.

REFERENCES

[1] A. Kamkin and A. Tatarnikov, MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors, proceedings of the 6th Spring/Summer Young Researchers’ Colloquium on Software Engineering (SYRCoSE 2012), 2012, pp. 64-69.

[2] A. Kamkin, T. Sergeeva, A. Tatarnikov and A. Utekhin, MicroTESK: An Extendable Framework for Test Program Generation, proceedings of the 7th Spring/Summer Young Researchers’ Colloquium on Software Engineering (SYRCoSE 2013), 2013, pp. 51-57.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov and A. Ziv, Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification, IEEE Design & Test of Computers, 2004, pp. 84-93.

[4] http://www.obsidiansoft.com/pdf/Datasheet.pdf

[5] L. Fournier, Y. Arbetman, and M. Levinger, “Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator: Application to the x86 Microprocessors Family,” Proc. Design Automation and Test in Europe (DATE 99), IEEE CS Press, 1999, pp. 434-441.

[6] R. Emek, I. Jaeger, Y. Katz and Y. Naveh, "Quality Improvement Methods for System-level Stimuli Generation", Proc. Computer Design: VLSI in Computers and Processors (ICCD 2004), IEEE CS Press, 2004, pp. 204-206.