The Government of the Russian Federation
Autonomous federal state institution
of higher professional education

National research university
"Higher school of economics"

School of Software Engineering
Software Management Department

MASTER'S DEGREE THESIS

Topic: «Architecture Model Based Microprocessor Test Generation Tool»

Student of class № 271mURPO
Andrei Tatarnikov

Scientific advisor
professor, doctor of phys.-math. sciences
Alexander K. Petrenko

Consultant
____

____

Moscow, 2013

Abstract

In the current industrial practice, test program generation and simulation is the main approach to system-level functional verification of microprocessors. Tremendous growth in complexity of modern microprocessor designs and tight time-to-market requirements make it an increasingly difficult task. Despite that fact that modern software market offers a range of powerful test program generation tools, development of functional tests still remains a laborious and time-demanding job, which makes functional verification the bottleneck of the microprocessor design cycle. A common problem of most such tools is that they were developed for particular microprocessor architecture and are hard to adapt to new designs. In fact, a tool typically has to be rewritten from scratch. A solution to this problem is to use architecture models that hold all knowledge about a specific microprocessor to configure a test generation tool. This work presents a test program generation tool called MicroTESK that uses formal specifications to describe the architecture of a design under verification. The specification is automatically translated to an architecture model that serves a basis for creating tests for the given microprocessor design. Such an approach helps keep to a minimum the effort required to configure a test generation tool for a particular microprocessor architecture, which significantly reduces the time required to develop functional tests.

Keywords—microprocessor design, test program generation, model-based testing, architecture description languages, test templates, random tests, constrained tests, combinatorial tests, directed tests.

Acknowledgements

I would like to express my gratitude to Prof. Alexander K. Petrenko, my scientific advisor and my manager at ISP RAS, and to Prof. Sergey M. Avdoshin, the head of the software Engineering school at NRU HSE, for a chance to work on this research project. This was the most interesting and challenging topic I could imagine. This work enriched my competence in the area of verification and significantly contributed to my further professional development. Also, special thanks to Alexander S. Kamkin, my team leader at ISP RAS, for his invaluable ideas and support. I really appreciate the chance to do interesting things together with good people! Finally, I would like to thank Prof. Rajat Moona from Indian Institute of Technology Kanpur who kindly shared with us materials on the Sim-nML architecture description language.

Contents

1. Introduction

As modern microprocessors are becoming more and more complex, the role of systematic activities for ensuring their correctness and reliability becomes critically important. These activities are called verification and testing. Verification is applied at the development stage and focuses on detecting logical faults in microprocessor designs that are represented by software models described in hardware description languages such as VHDL and Verilog. In other words, when we are talking about functional verification of microprocessors, we actually mean checking their models for correctness. As regards testing, which is also referred to as post-silicon validation, it is performed at the manufacturing stage to diagnose physical faults in integrated circuits. A common approach for both tasks is creation and execution of test programs that represent instruction sequence raising some events in the design under verification and optionally checking validity of the resulting microprocessor state.

Approaches to creating test programs and to performing functional testing have been a subject of research nearly since the invention of microprocessors. However, new approaches continue to emerge since increasing complexity constantly demands for new more efficient methods. Quality of testing is determined by the level of test coverage. Due to enormous ranges of states in modern microprocessors, creation of high-quality tests is a challenging task that requires a significant amount of time and effort. In fact, it is common that up to half of resources spent on microprocessor design is devoted to verification. In response to this issue, new methods aim to facilitate verification by using various techniques to automate this process. Approaches to automated test program generation can be divided into the following categories: random, combinatorial, template-based and model-based. The important point is that no single approach can be used as a "silver bullet" for all kinds of verification and testing tasks.

Consequently, in real-life practice different approaches are combined to complement each other.

Tools that create test programs for a given microprocessor architecture in an automated way are referred to as test program generation tools. A common drawback of such tools is that test generation logic is often tightly coupled with architecture-specific knowledge. This significantly complicates maintenance. In fact, when it is required to support new microprocessor architecture, a common solution is to rewrite the existing tool from scratch. No surprise, it increases the cost of microprocessor development and causes delays in the delivery schedule. Another important issue is that most of test generation tools available in the market are oriented on a limited set of test generation methods. As a result, when it is required to combine different techniques, verification engineers are forced to use a number of tools with different input and output formats. This causes certain inconvenience, since it might be problematic to integrate the tools and keep their configurations consistent. Such an approach works when different generation methods are used for connected, but independent tasks. For example, general functionality is tested by randomly generated programs, while tests for critical logic are generated with the help of advanced model-based techniques. However, when it comes to settling tightly dependent problems by means of two or more tools, it becomes a serious challenge as it might require deep knowledge of their internal architecture. The root of the problem is that each tool uses its own representation of the target design which is often hidden from others. As a consequence, knowledge about same design aspects is duplicated several times increasing the required maintenance effort proportionally to the number of integrated tools.

To overcome the described drawbacks, an efficient test generation tool should possess the following properties: reconfigurability and extendability. The former means an ability to easily switch to a new microprocessor design without having to modify the core functionality of the tool. The latter means flexibility of the tool architecture, which allows adding new functionality with minimal effort. In our case, it should be possible to integrate different generation techniques by installing corresponding extension components into the tool.

The thesis proposes an approach to the architecture of a test program generation tool that facilitates reconfiguring to new microprocessor architecture and integrating a wide range of test generation techniques. This approach is implemented in a tool called MicroTESK. The required flexibility is achieved by isolating logic responsible to specific tasks into independent components. To facilitate reconfiguring, knowledge about specific microprocessor architecture is encapsulated in an architecture model that is used by the architecture-independent core of the tool to generate test programs for the specified design. The architecture model includes an instruction set model and a coverage model (knowledge about situations to be covered by tests). Tools that follow this approach are usually called model-based test program generators. Typically, test cases for a microprocessor design are described manually in the form of test templates that represent an abstract description of a testing goal to be covered specified in terms of the instruction set model and the coverage model. Model-based test program generation is a time-proven approach implemented in industrial tools such as Genesys-Pro and RAVEN. However, creating a microprocessor's architecture model is rather difficult and requires special skills verification engineers usually lack for. To overcome this problem, MicroTESK makes use of architecture description languages (ADL), which are commonly used in the area of functional simulation, to specify the target architecture. The current version of the tool uses the Sim-nML language. This is a high-level formalism that has a format similar to pseudocode used in microprocessors' reference manuals to describe instruction semantics. Such a format is easy to maintain for verification engineers that lack programming skills. Usage of high-level specifications and automated translation of these specifications into architecture models make it easy to adapt the tool for new architectures or to reconfigure it for several revisions of the same design. To provide additional flexibility, the architecture of MicroTESK facilitates adding support for new generation techniques and integrating models of new design aspects (e.g. memory management and pipelining).

The tool is decomposed into two layers: core and extensions. The core includes functionality to model the instruction set and implementations of generation techniques based on knowledge about the instruction set. Such an approach is explained by the fact that every test program generator requires information about microprocessor instructions. Consequently, the instruction set model was chosen as an interface for integration of various generation methods. Extensions are responsible for solving custom generation and modeling tasks. For example, they may add facilities for modeling cache hierarchy and generating tests that would cover cache-related situations.

The architecture of MicroTESK and the ideas that lay behind the implemented approach are discussed in more detail further in this thesis. Chapter 2 contains an overview of existing test generation methods and tools. Chapters 3 and 4 introduce ADLs and provide a description of Sim-nML. Chapter 5 formulates technical tasks to be solved in the present work. In Chapter 6, a general description of MicroTESK's architecture is provided. Chapters 7 and 8 discuss the main parts of MicroTESK: modeling framework and testing framework respectively. Chapter 9 gives information about the results of trials of the tool with a Sim-nML specification of an ARM microprocessor. Chapter 10 lists publications dedicated to MicroTESK and conferences where it was presented or discussed. Finally, Chapter 11 concludes the thesis.

2. Existing methods and tools

The approach implemented in MicroTESK is based on a combination of well-known techniques coined from different sources. Over the last decades, a lot of industrial and academic research has been done into hardware verification methods. This chapter gives an overview of the most significant approaches and industrial tools and discusses their advantages and disadvantages.

The best known industrial test program generation tool is Genesys-Pro by IBM Research [5]. This is a model-based tool that operates with two kinds of knowledge: architectural model (includes an instruction set model and a coverage model) and test templates. Architecture models are created using high-level building blocks provided by the modeling framework included in the tool. Test templates represent an abstract description of verification scenarios specified in terms of the knowledge contained in the architectural model. Test templates allow defining preconditions for individual scenario instructions (e.g., boundary conditions, exceptions, cache hits/misses, etc.). For each precondition, the tool formulates a constraint satisfaction problem (CSP) and generates test data by solving this CSP. Unfortunately, there is no detailed information available on how to create architecture models for Genesys-Pro. It is known that modeling instructions that affect memory devices can be problematic. Therefore, there are reasons to think that Genesys-Pro is hardly reconfigurable if significant modification of memory devices' configuration is required.

Another popular industrial solution is RAVEN (Random Architecture Verification Engine) by Obsidian Software Inc (acquired by ARM) [6]. This tool generates fully random, semi-random or user-directed test programs for microprocessors. Like Genesys-Pro, it uses architecture models to configure the tool and test templates to specify user-directed scenarios. Architectural models are created by the tool vendor in cooperation with microprocessor manufacturers.

Configuration for custom designs can be done with the help of the generator construction set (GCS), a C++ API to the RAVEN core. Due to lack of information on this technology, it is hard to access how much effort it demands. However, creating an architecture model for RAVEN is unlikely to be an easy task for a verification engineer. Supposedly, it involves close interaction with the tool's developers, which is inconvenient and will inevitably lead to delays.

An interesting approach to modeling based on specifications in architecture description languages is discussed in the work of Prabhat Mishra and Nikil Dutt [7]. The idea is to use graph-based coverage models to generate functional tests. The model is automatically built from a specification in the EXPRESSION architecture description language [8]. Tests are generated in the following way: the tool processes the created model to extract test situations to be covered in a test program. This procedure is based on model checking. A test case is constructed as a counterexample for the negation of the target test situation.

Finally, Institute for System Programming of the Russian Academy of Sciences (ISPRAS) has already done some research dedicated to development of test program generation tools [1], [2], [17], [18], [19], [20], [28]. The present work summarizes the accumulated ideas and provides implementation for methods formulated in the earlier research works.

3. Architecture description languages

MicroTESK uses the Sim-nML [4] architecture description language to describe the architecture of the design under verification. This language is an extension of the nML [3] language, which facilitates creating simulation tools. nML was designed in the beginning of 1990s by Markus Freericks from Technische Universität Berlin. Now it is supported by Indian Institute of Technology Kanpur that proposed an extension to the original language called Sim-nML.

Before discussing facilities offered by Sim-nML in more detail, let us take a wide view of architecture description languages (ADL). First of all, it is important to understand what an architecture description language is. ADLs are high-level languages specifically designed to model microprocessor architectures. In contrast to hardware description languages such as Verilog and VHDL that describe the structure of electronic circuits in full details, they provide high-level specification of a microprocessor architecture (or so-called “programmer’s model”). ADLs facilitate extraction of information on instruction syntax and exploring behavioral properties of instructions. There are two criteria for classifying ADLs: content and objective. The content-oriented classification groups ADLs according to the nature of the information an ADL is aimed to describe. For instance, they can provide information on *behavioral* properties, on *structural* properties or *mixed* information. Structural ADLs are close to hardware description languages, they describe microprocessor units and mechanisms of their interaction. Structural ADLs can be used to synthesize hardware. Typically, there is no explicit specification of the instruction set, but it is usually possible to extract this information. The main disadvantage is that specifications in such ADLs contain a great number of details and creating specifications is a laborious task. An example of such language is MIMOLA [8]. Behavioral ADLs are designed to describe the instruction set of the target microprocessor. They specify the syntax and semantics of instructions, supported addressing modes and the structure of memory and registers. Such details as memory management and pipelining are usually skipped. Behavioral ADLs are convenient to create light-weight easy-to-maintain specifications. As a price for their simplicity, they cannot be used for hardware synthesis or for creating clock-accurate simulators. Languages belonging to this category include nML[3], Sim-nML [4] and ISDL [8]. Mixed ADLs combine traits of both structural and behavioral ADLs. They allow not only describing the instruction set, but also some details of the microprocessor's microarchitecture. The most common mixed ADLs are EXPRESSION [8] and LISA[8].

The objective-oriented classification is driven by the purpose of an ADL. Based on the objective, ADLs can be divided into the following categories: simulation-oriented, synthesis-oriented, compilation-oriented, and validation-oriented. There is no strict on-to-one correspondence between the two classifications.

To be successfully used to provide configuration for a test program generation tool, an ADL should satisfy some requirements. Notice that there is no ADL that was specially developed for this task. For this reason, a choice of formalism will involve some compromise. However, it is obvious that behavioral ADLs are more suitable. The list below contains requirements an ADL should satisfy. A suitable ADL should:

- be simple (minimum of low-level details);
- provide information on instruction syntax;
- describe instruction semantics;
- have public documentation and description of grammar;
- be extendable.

Several behavioral and mixed ADLs were considered as candidates to be used in MicroTESK. Eventually, Sim-nML was chosen as the most suitable. There are several ADLs that satisfy the formulated requirements. However, the main issue with most of them is that there is a little or no documentation available on these languages. For Sim-nML, Indian Institute of Technology Kanpur and Toulouse Research Institute in Information Technology (IRIT) provided manuals, grammar description and examples of specifications. For this reason Sim-nML was chosen to be used in MicroTESK.

4. Basics of Sim-nML

Sim-nML is a mixed ADL that is used for creating various simulators and disassemblers [4]. It is a high-level formalism targeted for describing arbitrary microprocessor architectures. Sim-nML works at the instruction set level hiding implementation details of the microprocessor design. Sim-nML is based on attribute grammar and represents a programmer's model that includes the following elements: register and memory definitions, supported addressing modes, syntax and semantics of instructions.

Sim-nML uses a hierarchical tree-like structure to describe an instruction set. Such a structure facilitates grouping related instructions and sharing their common parts. An instruction is described as a path in the tree from the root node to a leaf node. The set of all possible paths represents an instruction set. A node describes a primitive operation responsible for some task within an instruction. Nodes have attributes that can be shared with their parents. Actions performed by instructions are described as operations with registers and memory that represent bit vectors of arbitrary size.

A specification in Sim-nML starts with definitions of types and constants. For example, a type definition for a 32-bit word looks as follows:

```
let WORD_SIZE = 32
type word = card(WORD_SIZE)
```

Type definitions and constants can be used to describe registers and memory. In addition to registers and memory, it is also possible to define temporary variables, internal abstractions provided by Sim-nML to store intermediate results of operations. They do not correspond to any data storage in real hardware and do not save their data across instruction calls. Also, there is often a need to specify some properties of the described model. For this purpose, special constants are used. For example, the code below defines general-purpose registers, memory and a temporary variable. Also, there are two special constants that specify endianness and establish a correspondence between the general purpose register number 15 and program counter. Here is the code:

```
reg GPR[32, word]
mem M[2 ** 20, byte]
var carry[1,bit]
let byte_order = "little"
let PC = "GPR[15]"
```

As it has already been said, an instruction set is described as a tree of primitive operations. There two kinds of primitives: operations and addressing modes. Operations describe parts of instructions responsible for specific tasks and can be used as leaf and root nodes. Addressing modes are aimed to customize operations (for example, they encapsulate rules for accessing microprocessor resources). They can only be used as leaf nodes. For example, here are simplified examples of operation and addressing mode specifications:

```
mode REG(i: nibble)=R[i]
syntax = format("R%d", i)
image = format("01%4b", i)

op Add()
syntax = "add"
image = "00"
action = { DEST = SRC1 + SRC2; }
```

Operations and addressing modes have three standard attributes: syntax, image and action. The first two specify textual and binary syntax. The third describes semantics of the primitive. In addition, addressing modes have a return expression that enables them to be used as variables in various expressions. Attributes can be shared with parent primitives that refer to a given primitive.

Primitives are arranged into a tree using production rules. There are two kinds of production rules: AND rules and OR rules. AND rules specify parent-child relationships where a child primitive is described as a parameter of its parent. Here is an example of an AND rule:

```
op arith_inst(act: Add, op1: OPRND, op2: OPRND)
```

This is the header of the “arith_inst” operation that states that the “arith_inst” operation node has three child nodes: the “act” operation and the “op1” and “op2” addressing modes. The syntax of an operation header is similar to a function where parameter types specify the primitives the rule refers to. Parameter can be, in turn, parameterized with other primitives (they will be encapsulated behind attributes). For this reason child nodes represent independent instances that are accessed from their parent node via parameters. OR rules specify alternatives. This means that a group of primitives is united under some alias so that each of them can used when this alias is specified in an AND rule. An OR rule looks as follows:

```
op Add_sub_mov = Add | Sub | Mov
```

Figure 1 displays a tree path describing the “mov” instruction from an imaginary instruction set. This instruction copies data from one register to another. The root operation of the instruction is called “instruction”. According to Sim-nML conventions, there can be only one root operation. Usually the root operation is responsible for such common actions as increment of the program counter. The root operation is linked to the “Arithm” operation with the help of an AND rule. This operation describes a group of arithmetic operations. It is parameterized with the “Add_Mov_Sub” and “OPRND” primitives. Both of them are specified as OR rules. The first one describes arithmetic operations that can be performed by the

“Arithm” primitive while the second one specifies supported addressing modes. Dashed lines that connect OR-rules with their child primitives specify possible alternative paths. Instructions are identified by the terminal operation node of the path (in this example, it is the “Mov” node). An important note is that, to avoid ambiguity, nodes can have only one child operation.

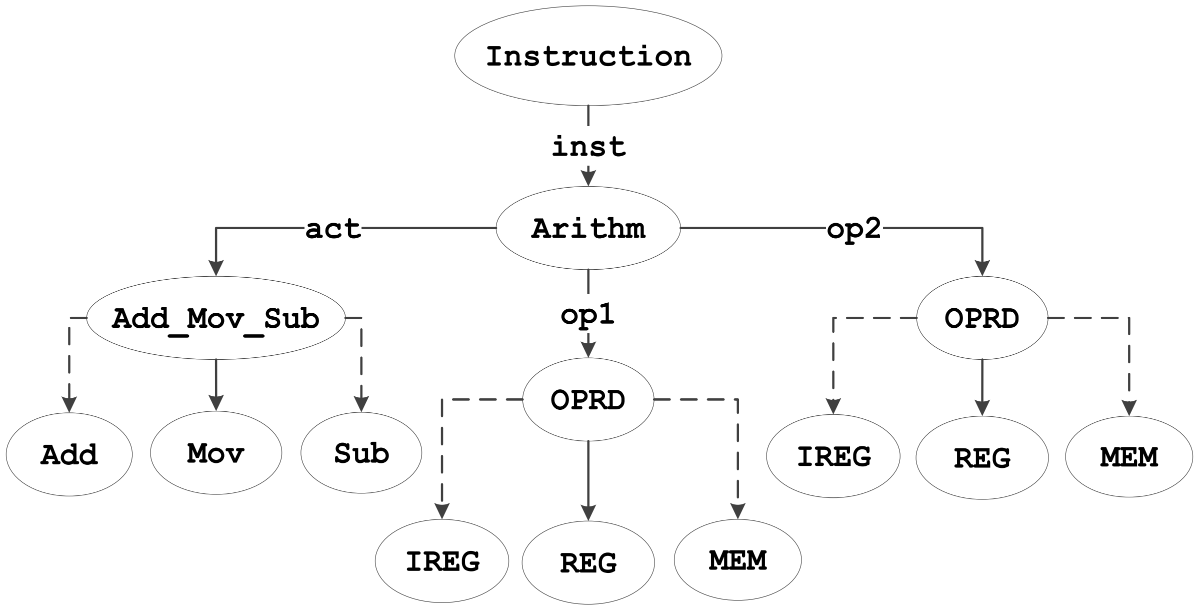

Figure 1. Operation tree for the Mov instruction

The syntax of Sim-nML resembles the syntax of the pseudocode used in microprocessor architecture manuals to describe instruction semantics. For example, here is the description of instruction ADD from MIPS64 manual [9]:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
    UNPREDICTABLE
endif
temp ← GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
    SignalException(IntegerOverflow)
else
    GPR[rd] ← sign_extend(temp31..0)
endif
```

Such a description can be translated to Sim-nML with minimal effort. Providing that all needed data types, resources and operations describing common functionality of instructions have already been specified, the specification of the ADD instruction (or, to be more precise, the terminal operation that distinguishes it from other similar instructions) will look as follows:

```
op ADD(rd: GPR, rs: GPR, rt: GPR)
action = {
    if (NotWordValue(rs) || NotWordValue(rt)) then
        UNPREDICTABLE();
    endif;
    tmp_word = rs<31..31>::rs<31..0> + rs<31..31>::rt<31..0>;
    if(tmp_word<32..32> != tmp_word<31..31>) then
        SignalException("IntegerOverflow");
    else
        rd = sign_extend(tmp_word<31..0>);
    endif;
}
```

As we can see, describing an instruction based on an instruction set manual is a relatively easy task that can be performed by a verification engineer who does not have significant programming skills.

Describing all features of Sim-nML in full detail is out of scope of this work. Detailed descriptions of nML and Sim-nML can be found in works [3] and [4] respectively.

5. Research and development task description

The goal of the present work is to propose an approach to the architecture of a test program generation tool that facilitates reconfiguring the tool to new microprocessor designs and simplifies adding support for new test generation techniques. The proposed approach will be implemented in test program generation tool MicroTESK. Here is the list of tasks that should be implemented:

1. Design the architecture of MicroTESK. This includes decomposing the tool into subsystems and designing interfaces that will describe mechanisms of their interaction.
2. Implement the model API (a library of interfaces and classes that will be used as building blocks for the model).
3. Implement the Sim-nML translator. The translator processes a specification in Sim-nML and generates an architecture model in Java on a basis of the model API.
4. Implement the test generation engine and API for creating various generators. The generation engine processes test templates and generates tests for the specified architecture. It includes a set of generator components responsible for specific tasks implemented basing on the API.
5. Implement test sequence generators. During the first stage of test generation, concrete sequences are produced on a basis of an abstract description provided in test templates. Sequence generators implement different techniques of producing instruction sequences.
6. Implement test data generators. Arguments of some instructions may not be specified explicitly in test templates. Instead of being specified as concrete constant values, they can be generated at random or calculated

by solving a CSP. Calculation of such parameters is performed at the final state of test generation by test data generators that implement specific techniques of producing test data.

7. Implement constraint solver API. MicroTESK uses external constraint solver engines to generate test data. To interact with them, special API should be implemented.
8. Create a specification of a real microprocessor. To demonstrate the implemented solution and to assess its effectiveness, a Sim-nML specification of a real (or close to real) microprocessor architecture should be provided.
9. Create examples of test templates for the specified microprocessor architecture.

MicroTESK is an open-source tool developed at ISPRAS. The main development instruments are Java, Eclipse, ANTLR and JRuby.

6. Architecture of MicroTESK

MicroTESK performs two primary tasks: (1) synthesis of architecture models on a basis of design specifications and (2) generation of test programs for the given architecture model from test templates. Consequently, the tool is decomposed into two loosely coupled features: (1) modeling framework and (2) testing framework. Generally speaking, an architecture model serves as output data for the former and as input data for the latter. A high-level scheme of MicroTESK's architecture is displayed in Figure 1.

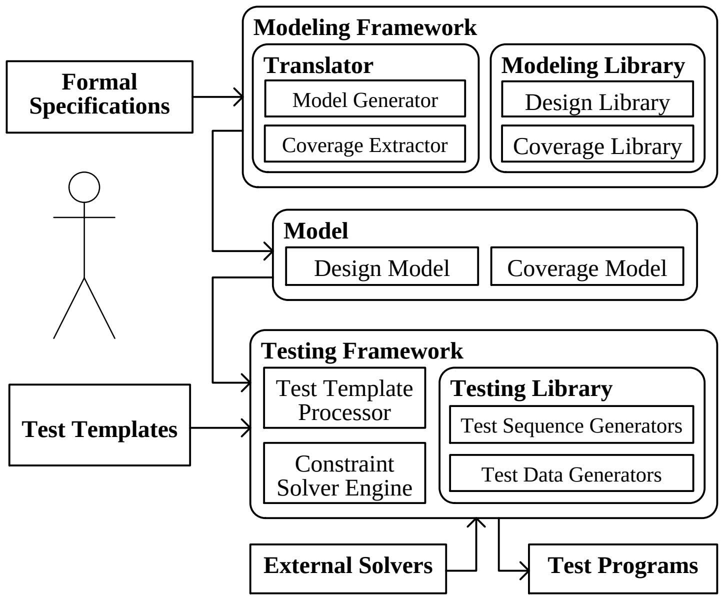

Figure 2. General structure of MicroTESK

The modeling framework is responsible for building an architecture model on a basis of provided formal specifications. It includes two main components: the translator and the modeling library (that contains design and coverage libraries). The former processes specifications and generates a model in Java and the latter provides building blocks for the model. The testing framework generates tests on a basis of test templates and the model provided by the modeling framework. Its components are as follows: a test template processor (processes test templates to generate test programs), a testing library (contains test sequence generators and test data generators used by the test template processor) and a constraint solver engine (organizes interaction between test data generators and external SMT solvers).

Test program generation with MicroTESK is performed in several stages that use different components. Here are the generation stages:

- A verification engineer provides an ADL specification of the target microprocessor architecture based on design documentation.
- The translator of the modeling framework parses the specification and builds an architecture model in Java using building blocks provided by the modeling library.
- The verification engineer creates test templates for test cases to be generated. Test templates are described in terms of the architecture model.
- The test template processor of the testing framework builds an internal representation of the provided test template and processes with generators from the testing library.
- The test sequence generators of the testing library produce a series of abstract test programs that specify instruction sequences to be placed in generated test cases, but do not specify concrete test data for instruction calls where they has not been explicitly specified as constant values.
- The test data generators of the testing library generate all required testing data with the help of constraint solver engine and produce a series of concrete test program (expressed as some internal representation).
- The test template processor simulates the execution of concrete test programs with the architecture model to update the model’s internal state and generates source code of test programs.

Components that are involved in test generation are described in subsequent sections.

7. Modeling framework

The role of the modeling framework is to prepare the environment for test generation. It provides functionality creating a microprocessor model basing on formal specifications. As it has already been said, there two principal components: the translator and the modeling library. Before discussing then in detail, let us have a closer look at the generated microprocessor model.

7.1 Microprocessor model

The microprocessor model consists of the design model and the coverage model. Information provided by both models serves as a basis for describing test cases. Unified interfaces of the model allow applying various generation techniques implemented in MicroTESK to different microprocessor design without having to modify the tool's core. The conceptual scheme of a microprocessor model is displayed in Figure 3. Interaction with the model is performed using three abstractions: (1) meta information provider, (2) model state observer and (3) instruction call configurator. They encapsulate details of the design model and the coverage model and make them appear as a whole. In fact, both models are tightly coupled, but describe different properties of a microprocessor. The coverage model is built on top of the design model and summarizes its behavioral properties.

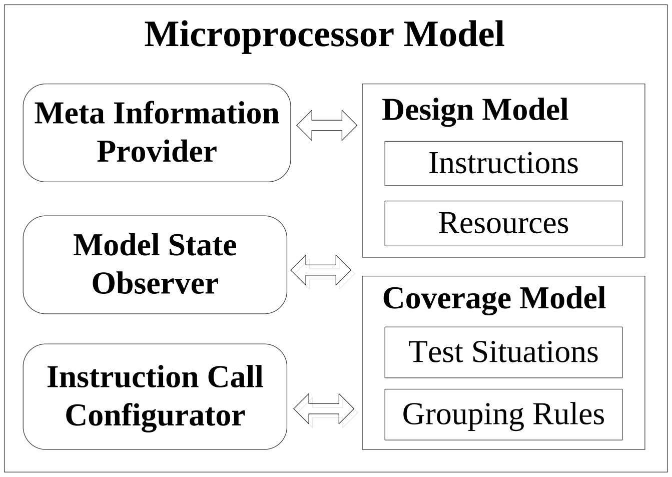

Figure 3. Conceptual scheme of a microprocessor model

The main responsibility of the design model is tracking the state of the microprocessor during test generation. Information on the current state is needed to create self-checking tests and to generate test data on a basis of constraints formulated in terms of the current state. The design model allows simulating execution of microprocessor instructions and monitoring microprocessor resources (registers, memory). Also, the design model is responsible for producing textual and binary code of simulated instruction calls that will be inserted in test programs. The instruction set model serves as a core of the design model (information on instruction syntax is essential for all kinds of test generation techniques). In the current version of MicroTESK, the design model contains only an instruction set model based on Sim-nML specifications. However, the design model has extension points that can be used to integrate models of microarchitecture elements such as memory management unit.

Interaction with the design model is performed via the model state observer and instruction call configurator. Both interfaces are independent of the model configuration (instruction set, registers, memory configuration and other properties). To identify elements of the model, each design element provides meta information which is available through the meta information provider. This information serves as a basis for specifying instruction calls and requests for information on the model state. This allows working with different models in a uniform way and adding support for modeling new elements without having to modify the interfaces.

The coverage model contains information about behavioral properties of the target design. It describes test situations and rules for grouping similar instructions. This information serves as a basis for creating test cases. Test situations are typically described as constraints expressed in terms of instruction parameters and the model state.

From a programmer's point of view, generated microprocessor models are represented by a hierarchy of objects hidden behind public interfaces. The public part of the model includes objects that implement the three main services of the model (namely, (1) meta-information provider, (2) model state observer and (3) instruction call configurator). These abstractions interact with the internal part to perform their tasks. The internal part of the model contains the following entities:

- A model of memory resources (register banks, memory lines, temporary variable stores, internal flags). Information it stores can be read and written using the state observer. To identify specific resources, the entity metadata describing its contents. The metadata is aggregated by the microprocessor model's metadata provider. Inside the model, operations with resources can be performed by the addressing modes and operation primitives that have full access to them.
- Addressing mode models. Provide implementations for the rules for accessing memory resources. Allow reading/writing data in a uniform way from/to various sources. Used as input parameters of instructions. To be able to be dynamically instantiated and initialized with different input parameters, addressing modes provide special builders that allow users of the model to create required objects. Like other entities, provide metadata that describe the format of their input parameters. Addressing modes described by OR-rules (a set of alternatives) aggregate metadata and builders of other addressing modes allowing choosing at runtime which object should be created.
- Operation models. Represent components that perform smaller parts of instruction tasks (contain parts of instruction logic and syntax description). Operations aggregate other operations and addressing modes to build fully functional instructions.

- Instruction descriptions. Implement the rules of grouping operations and addressing modes into instructions. The name of the terminal operation and the list of aggregated addressing modes form the instruction signature. Instructions provide metadata that describes their signatures and builders that help set up instruction calls.
- An instruction set description. A container that holds the list of supported instructions. It aggregates metadata of stored instructions and provides facilities for accessing specific instructions.
- Test situation descriptions. Specify conditions that cause certain events to occur. Test situations are associated with instructions raising the described events. The test situation entities provide metadata for describing their signatures and builders for creating and initializing them.

These entities are hidden behind public interfaces that make a model appear as a whole. All metadata provided by separate entities is united in a hierarchical structure by the meta-information provider. It provides descriptions of memory resources and instructions (including their parameters and test situations associated with them). Instructions can be grouped according to some characteristics. The model state observer provides access to memory resources using a unified interface that uses metadata to identify recourses. The instruction call configuration provides access to call builders of supported instructions. This allows the client code to create an instruction call, specify needed parameters (using argument builders and addressing mode builders) and specify conditions (test situations) that should satisfied by instruction arguments (if they were not set up explicitly). Figure 4 shows the internal structure of a microprocessor model and links between the entities it contains. The entities are implemented and organized into a model with the help of library classes from the modeling library discussed later in this work.

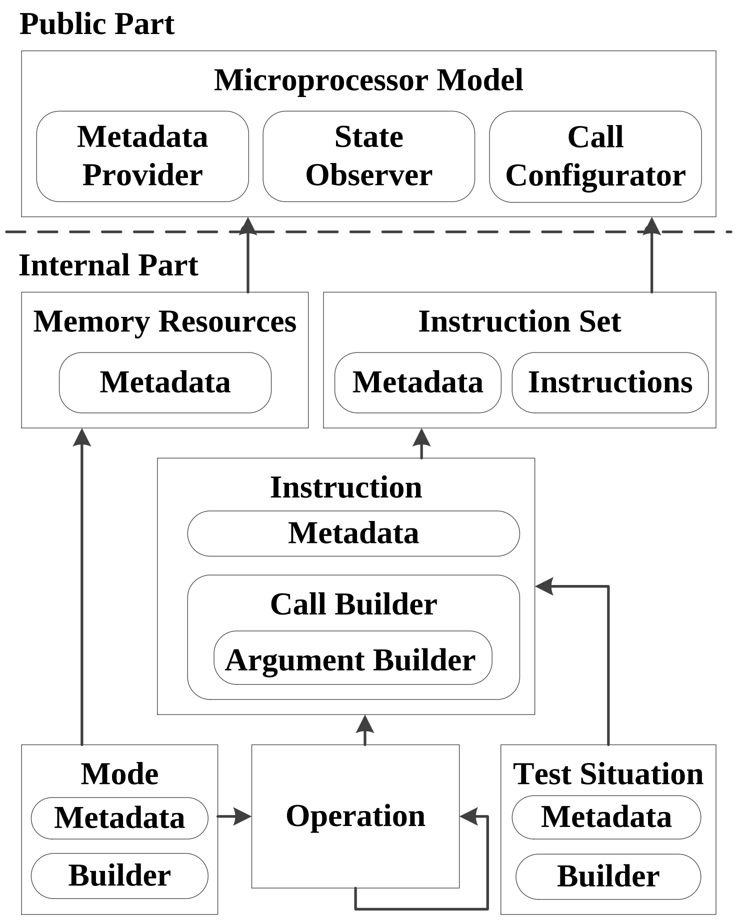

Figure 4. Internal structure of a microprocessor model

7.2 Translator

The translator processes a microprocessor specification in Sim-nML to produce a model. The translator consists of two parts: model generator and coverage extractor. The former is responsible for building a design model and the latter extracts coverage information and builds the coverage model. The structure of the translator is shown in Figure 5.

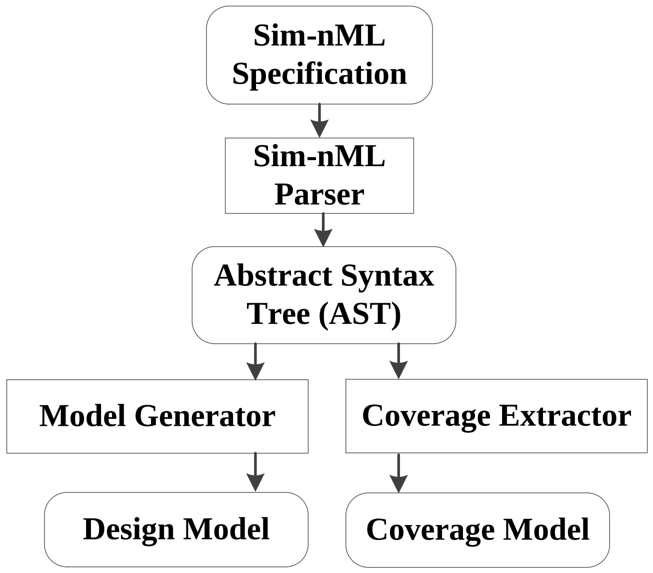

Figure 5. Structure of the translator

The translator is implemented using the ANTLR parser generator [10], [11]. The translator consists of a front end and two back ends. The front end is represented by a lexer and a parser created with the help of ANTLR on a basis of Sim-nML grammar. Their job is to build an internal representation (AST) for the processed Sim-nML specification. The back ends are the model generator and the coverage extractor. They analyze the AST and build the design model and the coverage model. The back ends consist of the following parts: (1) an AST walker, (2) builders of intermediate representation and (3) code generators. The AST walker generated by ANTLR traverses the tree to collect needed information. It uses special builder objects to create intermediate representation of the model which is represented as a table of primitives. The table is then processed with code generators responsible for generation of specific model classes. Here is the table of primitives extracted from Sim-nML specifications by the model builder:

Primitive	*Description*
Constants (let expressions)	Described by let constructs. Represent statically calculated constants that can be used in all parts of

	the specification.
Labels (or let labels)	Described by let constructs. Their role is to specify properties of the model (e.g. program counter register, endianness, etc.). They are not referred by other primitives in the specification.
Type definitions	Described by type constructs. Specify data types that are represented by bit vectors of the specified size. In addition to size, data types use type identifiers that set up how to treat these data types (as signed/unsigned integers, booleans, floating-point numbers, etc.).
Memory resources (registers, memory lines, temporary variables)	Described by reg, mem and var constructs. Specify memory resources as arrays of locations that have specified type and length.
Addressing modes	Described by mode constructs. Specify logic of accessing microprocessor memory resources. Have the following attributes: return expression (logic of resource access), syntax (textual format of the primitive), image (binary format of the primitive) and action (additional logic that can be executed from operations that use the given addressing mode).
Operations	Described by op constructs. Specify operations that build up instructions. Have the same attributes as

	modes, but do not provide a return expression. Operations semantics is described by the action attribute.
Instructions	Produced by analyzing tables of other primitives. Represent a composite object that contains a tree of operations and addressing modes (references to entries in other tables).

Table 1. Information extracted from Sim-nML specifications

Code generation is performed using string templates that facilitate creating complex Java classes [12]. A generator for each primitive kind consists of a generation class and a string template. String templates specify component classes and interfaces provided by the modeling library that will be used in generated classes.

In other words, to perform translation of each primitive kind, the following features are provided (each is responsible for accomplishing different steps in the translation chain):

1. Token rules (lexer rules) for splitting the input file into a sequence of tokens. Described in the ATNLR DSL for lexer grammars.
2. Parser rules for building an AST from tokens provided by the previous stage. Described in the ATNLR DSL for parser grammars.
3. Tree walker rules traversing the AST. They describe the recursive-descent tree traversal algorithm that collects information on primitives and uses it to build their intermediate representation.
4. Classes for building and storing the intermediate representation of the model primitives. Builder classes are used by the AST walker to create

the intermediate representation that will be used by generators to create the model code.

5. Templates for classes to be generated and code for initializing these templates with parameters extracted from the Sim-nML specification. Described using the DSL and Java API classes provided by the String Template library.

The coverage library uses the same approach to create classes for test situations and grouping rules.

7.3 Modeling library

The modeling library provides base classes for creating microprocessor models. From a programmer's point of view, the design library and the coverage library represent a single library. The division is rather conceptual and is aimed to highlight that classes from these libraries are used by different code generators to create model classes serving different purposes. The model library consists of smaller libraries that provide building blocks for describing features of the model at different levels of detail. Higher-level libraries are created on a basis of lower-level libraries. The table below lists the main libraries from the lowest level of detail to the highest.

Library	Description
Raw data library	This is a general-purpose library that provides classes for storing bit vectors. Includes code for concatenating bit vectors and creating masks that allow treating a single bit vector as a set of smaller bit vectors. Also, provides functionality for converting bit vectors to/from other data formats (integers, strings, byte arrays, etc).
Data type library	Provides classes for describing data types used in Sim-

	nML. Data is stored in bit vector classes provided by the raw data library. The data type library contains implementations for all operations supported by the described types.
Memory library	Contains classes for describing memory resources such as register files, memory lines and temporary variable stores. A single register or a variable is described an abstract entity called a location. The library also provides a connection point for integration with the memory management unit model (describing caches and address translation mechanism).
Exception library	A model can raise an exception. There two kinds of situations when it can occur: (1) an attempt to configure an instruction call or to request the model state using invalid parameters (configuration exception) and (2) incorrect semantics of an instruction that brings the model to an invalid state during simulation of its execution (simulation exception). The library provides classes for all possible exceptions of both kinds.
Instruction library	Contains interfaces and abstract base classes to be inherited by classes implementing instructions. These interfaces and classes are independent of specific ADLs and can be used to build an instruction set model described using different formalisms. The library describes the following abstractions: an instruction set, an instruction, an instruction call, an addressing mode, and instruction call builder, an instruction argument builder and an addressing mode builder.
Meta information library	Provides interfaces and classes for describing metadata for entities contained in a model (such as instructions,

	addressing modes, memory resources and test situations).
State observer library	Contains classes and interfaces that provide access to the internal state of the model. The allow requesting values stored in registers and memory and also status flags that indicate some “interesting” states of the model.
Sim-nML instruction library	Provides base classes for instructions built on a basis of Sim-nML specifications. Includes classes for such Sim-nML primitives as operations and addressing modes and their builders.
Model library	Provides bases classes for microprocessor model. They aggregate all lower-level components and organize access to them using such abstractions as meta data provider, model state observer and instruction call configurator.
Test situation library	Provides base classes for test situations extracted from the specification by the coverage analyzer. Includes classes that help describe preconditions as constrains that can be solved by the constraint solver engine.

Table 2. Libraries of the modeling framework

Microprocessor models created by the translator include the following class types (all of them extensively reuse library classes so that the generated code contains minimal functionality which is unique for the given model):

- *Model*. The main class of the model. It is inherited from the SimnMLProcessorModel library class. It aggregates the instruction set and memory resources and provides access to them via the IModel interface.

- *Shared*. Aggregates all constant, type, memory resource definitions. In addition, it holds internal statuses of the model and labels that associate specific memory recourses with their aliases. The Type, MemoryBase, Status and Label classes for type definitions, memory resources, internal statuses and labels respectively.
- *ISA*. This class it inherited from the InstructionSet class. Its purpose is to aggregate a collection of instructions.
- *Instruction*. Such a class holds all information about a specific instruction. It is inherited from the InstructionBase class. The main responsibility is to create an instruction call as a hierarchy of operations and addressing modes initialized with customer parameters.
- *Operation*. Implements the "op" abstraction of Sim-nML. Inherited from the Operation library class. Contains information on syntax and semantics of an operation that represents a part of an instruction.
- *Mode*. Implements the "mode" abstraction of Sim-nML that describes the logic of addressing modes. Inherited from the AddressingMode library class.
- *Situation*. Describes a particular test situation. Inherited from the Situation library class.

The modeling library was designed to simplify as possible creation of microprocessor models. The library classes help of create mode code in Java that looks as similar as possible to Sim-nML specifications, encapsulating all low-level details. This facilitates debugging of the model and finding bugs in the specification.

8. Testing framework

The job of the testing framework is to generate test programs on a basis of test templates provided by a verification engineer. Test templates represent an abstract description of a test case specified in a special template description language derived from Ruby[15]. Test templates are processed with the test template processor that utilizes engines from the testing library to produce test programs. There are two kinds of generation engines: (1) test data generators and (2) test sequence generators. Test program generation is performed in the following steps:

1. The test template processor applies test sequence generators to produce an abstract test program (i.e., a sequence of objects describing test calls that use test situations to specify preconditions for input parameters instead of concrete values).
2. The test template processor uses test data generators to create input data for instruction calls which will satisfy the preconditions formulated in test situations.
3. The test template processor creates an initialization section of the test program, which initializes registers and memory with data produced by the previous stage. Thus, the resulting output is represented by a sequence of instructions calls that have all required parameters unambiguously specified.
4. The framework generates source code of the test program (including the initialization section) and updates the model state by simulating execution of the generation instruction call sequence.

Components involved in test program generation are described in more detail in the subsequent subsections.

8.1 Test templates

Before discussing test generation engines, it is necessary to have a clear understanding of such a notion as test templates. Generally speaking, a test template is a high-level description of a test program to be generated. It uses a special notation that allows describing an instruction sequence in an abstract way without having to specify concrete instruction arguments or even concrete instructions in test cases. For instance, it is possible to define input data in terms of preconditions and to use groups of interchangeable instructions (optionally, with probability distributions of their occurrences) instead of specific instructions. Concrete data and instructions to be used in test programs will be chosen by the engine during test generation depending on input parameters and the state of the microprocessor model. Another important feature of test templates is the ability to describe instruction sequences that will be built using random, combinatorial or other user-defined algorithms. Also, it is possible to merge instruction sequences produced using different generation methods.

The test template description language represents a high-level scripting language (the current version uses Ruby[15]) extended with libraries that facilitate describing test cases. Such an approach helps keep to minimum the required learning effort.

Features provided by the language and the libraries serve the following main purposes:

- Select generation methods (i.e. chose sequencing algorithms, data generation engines, etc.).
- Configure instruction calls (with input values or using preconditions).
- Specify dependencies between instruction calls.
- Organize loops and conditional generation.

- Insert validity checks.
- Provide the infrastructure for creating complex tests.

The specified duties are common for all kinds of tests based on different types of testing knowledge and exploiting various test generation techniques. Since MicroTESK can be extended with new test generation methods, the test template description language is designed to be flexible to avoid modifying existing logic when a new feature is added.

Test templates are created using special test template API in Ruby that provides base classes for test templates and functionality for organizing test templates into groups. A test template is represented by a class that has public methods for initialization, finalization and test case execution. A creator of a test case only needs to provide implementations of those methods (all logic responsible for test generation is encapsulated in the base API class). The code below illustrates this approach:

```
class MyTestCase < Template

 def initialize
  super
  @is_executable = yes
 end

 def pre
  // Place your initialization code here
 end

 def post
  // Place your finalization code here
 end

 def run
  // Place your test case here
 end

end
```

The methods contain code that describes instruction sequences to be generated. Template code represents a hierarchical structure of test sequence blocks. They hold a set of instructions or nested blocks and specify what test sequence generator will be used to produce a test sequence. Instruction sequences for nested blocks are produced by recursively applying corresponding generation engines and merging blocks of the same level with the engine specified by the root block. The specification of a sequence block consists of a header and a body. The role of the header is to specify the sequence generation engine to be applied and parameters used to configure it. Parameter sets can vary for different engines. In order to keep the format of block headers uniform, parameters are specified as key-value pairs. Here is an example of a test template that has two nested sequence blocks using different generation engines with different sets of parameters:

```
# Test Sequence Block
block (: combine => ”product”,
        : compose => ”random”)
{
  # Nested Block A
  block (: engine => ”random”,
          : length => 3,
          : count  => 2)
  {
    add r(2), r(0), r(1)
    sub r(3), r(1), r(2)
    mul r(4), r(2), r(3)
    div  r(5),  r(3), r(2)
  }

  # Nested Block B
  block (: engine => ”permutate” )
  {
    ld r(0), r(4)
    st r(1), r(5)
  }
}
```

From a conceptual point of view, a test sequence block represents a specification of a single testing task. This task can have its own preconditions, postconditions and invariants that should be applied to the whole specified instruction sequence. Consequently, some generation engines may use block-level test situations and constraints. For such generation techniques, it is possible to assign test situations to whole blocks via parameters in the block header.

The body of a block describes instruction calls to be inserted into the generated sequence. Please note that the final order of instruction calls is determined by applied sequence generators and may not be the same as specified in the block. A template description of an individual instruction call includes the following properties:

- *Instruction identifier*. It can be the name of a specific single instruction, the name of an instruction group or a set of instruction names. The probabilities of occurrence of instructions in a set are assumed equal unless corresponding probability distributions are explicitly specified.
- *Addressing modes*. Sources of input data and destinations for output data of instructions are specified using addressing modes that provide access to microprocessor's memory resources. Each instruction argument can have one or several addressing modes associated with it. Test templates allow using a concrete addressing mode or randomly choosing an addressing mode from the specified set. Thus, it is possible to cover scenarios that involve access to different resources using the same test template.
- *Instruction arguments*. Input and output data are accessed by instructions via addressing modes parameterized with some constant values. These values can represent immediate values (for instance, an address or constant) or identify particular registers. Test templates allow specifying them in three ways: as concrete constant values (1), as random values (2)

and as constraints based on some preconditions (3). Such an approach facilitates covering a wide range of testing goals.

- *Test situations.* Specify a certain coverage goal (i.e. a logic branch executed under certain conditions). A test situation or a set of test situations can be linked to an instruction call. In this case, the generation engine produces input data that satisfy conditions necessary for the situation to occur and inserts corresponding initialization code in beginning of the generated instruction sequence.
- *Dependencies.* Instructions can share input and output data as well as other properties assigned when the template is being processed by the generation engine. The test template language allows establishing such links by using variables.

Test cases may require insertion of different instruction sequences into a test program depending on some conditions or repetition of some instruction sequences. The test template language provides constructs to support conditional branching and loops (in fact, they are inherited from Ruby). Conditions can be based on generated test data or on the state of the microprocessor model (the test template description language provides special facilities for querying information on the model state).

Below, there is an example of a template specification of a simple test case for the ALU of an ARM microprocessor.

```
(1..10).each do { |i|
   eor blank, setsOff, r(0), r(0), register0
   add blank, setsoff, reg(2), reg(1), register0   do normal end
   mov_immediate blank, setsoff, reg(1), immediate(i)
   add blank, setsoff, reg(1), reg(4), register4   do random end
   sub blank, setsoff, reg(3), reg(2), register1    do overflow end
   }
end
```

The example demonstrates the use of test situations ("normal", "overflow", "random") that specify preconditions for input parameters. Also, it demonstrates how the for loop from the Ruby language can be used in test templates.

One more important feature of test templates that is worth noticing is support for creating self-checking tests. Test templates can include checks for correctness of the microprocessor state. Such a check represents code that performs comparison of values stored in the specified register or memory address with expected values and terminates the program if they do not match. Termination is often performed as a control transfer to some address that stores code responsible for program termination and dumping the results. Test templates help automate the task of generating assembler code for validity checks, thus, reducing the effort needed to create self-checking tests.

8.2 Test template processor

The role of the test templates processor is to control the process of test generation. From a technical point of view, it is a runtime environment where scripts of test templates can be executed producing a test program. Generally speaking, it is set of Ruby libraries that serve as a basis for creating test templates and that provide a means of interaction with the microprocessor model and with the testing library. Basically, it performs its duties in the following main steps:

- Creates wrappers for the elements of the microprocessor model basing on provided meta-information (this includes instructions, their addressing modes, test situations, memory resources, etc). In other words, it creates implementations for such functions as “add”, “sub”, “reg” and other similar primitives used in test templates to describe instruction calls. Technically, to make these features available to test templates, it dynamically defines corresponding methods of the Template class.

Method bodies are defined as lambda functions parameterized with data taken from model’s meta-information.

- Runs the script of a test template to build the internal representation of the test sequence block hierarchy it describes. The script uses factories provided by the testing library to create required objects. To enable using Java libraries in Ruby scripts, MicroTESK uses the JRuby engine [16] to execute scripts.
- Processes the test sequence block hierarchy with test sequence generators from the testing library to create an abstract test program (a set of sequences of abstract instruction calls). The output is called an abstract test program because, at this step, the order of instruction calls is fixed while the values of their arguments may not be known yet (such instruction call descriptions are referred to as abstract test calls).
- Processes the abstract test program with test data generators to generate instruction call arguments that satisfy conditions formulated in test situations associated with specified instruction calls. An instruction call with fixed argument values is called a concrete call and a sequence of such calls is referred to as a concrete test program. Instructions that use arguments stored in registers or the main memory require initialization code to be executed to assign the resources values produced by data generators. For such instructions, corresponding initializing instructions calls are inserted in the beginning of the instruction call sequence (this part of the sequence is called an initialization section). An important note is that test data generation is performed sequence by sequence (if a concrete program consists of several instruction sequences) in order to be able to update the model state and to generate data of a basis of the updated state. A processed sequence is passed to the next step and only

when the next step is done the processor starts processing the next sequence.

- Simulates the execution of concrete call sequence to update the model state and writes textual representation of instruction calls to a text file. Then the control is transferred to the previous step. When all concrete call sequences have been processed, the resulting text file represents a generated test program that can be executed to check the validity of the microprocessor design.

The scheme of test template processing is displayed in Figure 6.

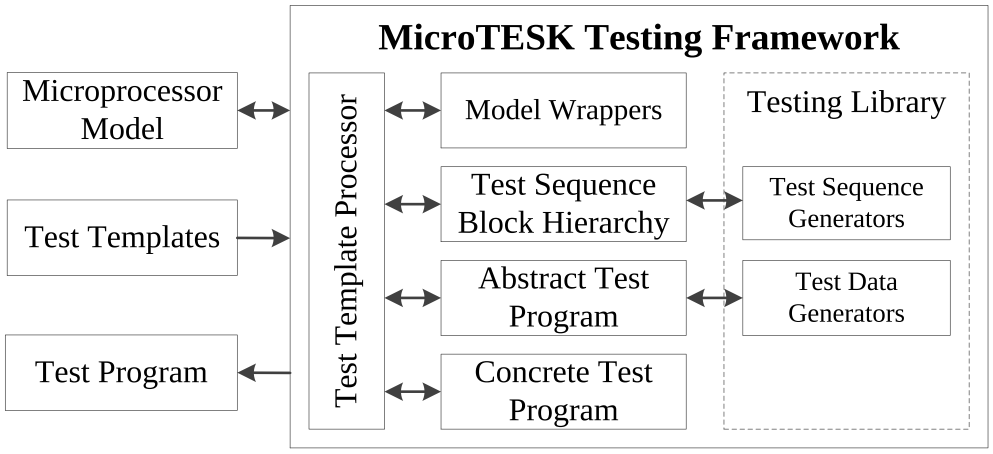

Figure 6. Scheme of test template processing

8.3 Test sequence generators

Test sequence generators process test sequence blocks to produce sequences of abstract instruction calls. The testing library contains a table of test sequence generators implementing a uniform interface. According to the identifier specified in the header of a test sequence block in a test template, a proper generator is selected to generate a sequence. Such architecture makes it easy to extend the testing library with new sequence generation techniques. To do this, a corresponding component implementing the sequence generator interface should be added to the table.

In the current implementation, test sequence generators are hidden behind the interface of test sequence blocks. In fact, when the test template processor is building a hierarchy of test sequence blocks the sequence generators are applied to the blocks being built to produce corresponding sequences. In turn, each test sequence block has a public method that returns an iterator for the collection of sequences generated for this block by a corresponding sequence generator. In the simplest case, the generator produces a single test sequence for a single test sequence block. For nested blocks, generated sequences are united in a recursive manner. To accomplish this task, each non-terminal block should specify the following strategies: (1) the combination strategy and (2) the composition strategy. The first describes how to combine sequences returned by iterators of nested blocks. The second describes how several sequences can be merged together. Thus, the testing library includes two types of test sequence generators used together to handle nested blocks: combinators and compositors. Combinators produce combinations of the nested test sequences, while compositors merge those sequences into a single one.

The testing library includes several standard combinators and compositors (if needed, the library can also be extended with custom ones). The current implementation provides the following standard combinators:

1. Random combinator. Produces a number of random combinations of the results returned by sequence generators of nested blocks.
2. Product combinator. Creates all possible combinations of the test sequences produced by nested blocks.
3. Diagonal combinator. Synchronously requests sequence iterators of nested blocks and joins the returned results.

Here are standard compositors provided by the testing library:

1. Random compositor. Randomly mixes test sequences retuned by nested blocks.
2. Catenation compositor. Catenates nested test sequences.
3. Nesting compositor. Embeds nested test sequences one into another.

In addition to standard components, it is possible to create custom test sequence generators, combinators and compositors, add them to the testing library and invoke them from test templates.

8.4 Test data generators

Test data generators are used by the template processor to translate abstract instruction calls into concrete instruction calls that can be simulated by the microprocessor model. As it has already been said, instruction calls described in test templates are not required to have explicitly specified arguments. Instead, it is possible to use random values, constraint expressions or test situations. For example, the following code snippet specifies the ADD instruction from the ARM ISA invoked for two random registers and a random immediate value:

```
add_immediate blank, setsoff, _, _, _
```

The ‘_’ identifiers are used to specify random arguments. The scope of random values can be limited with constrains. To create directed tests, instruction calls can be attributed with test situations describing conditions that should be satisfied by arguments to cause a certain event. For example, the following line of code states that the addition of two values stored in general purpose registers should cause an integer overflow:

```
add equalcond, setsoff, reg(1), reg(2), register0 do overflow end
```

Test data generators are responsible for generating appropriate arguments values and creating code that will initialize corresponding memory resources with these values (if needed). For different kinds of test situations, different data generators can be used. Most commonly, test situations are described as constraints on the internal state of the model and instruction argument values. For constraint-based test situations, MicroTESK used a test data generator that interacts with external SMT solvers to produce test data. The component responsible for this interaction is called constraint solver engine (it will be discussed in the next subsection). Figure 7 illustrates the scheme of interaction between components involved in creation of a concrete instruction call and corresponding initialization code.

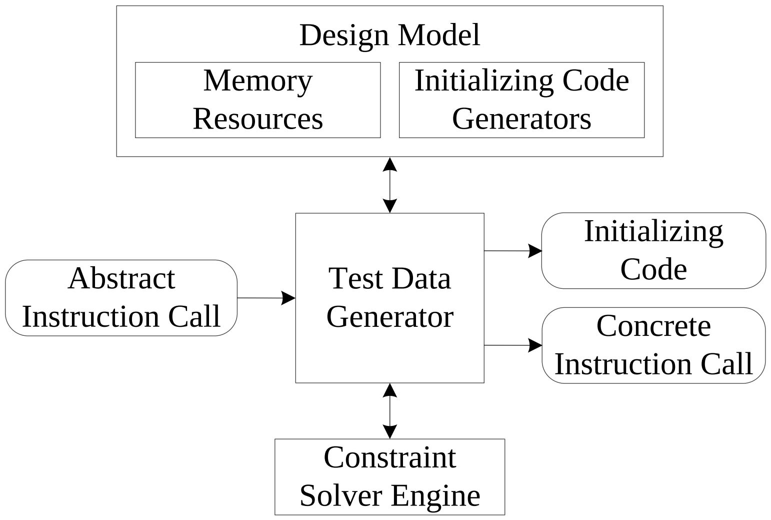

Figure 7. Scheme of test data generation

The general algorithm of generating data and initializing code on a basis of a provided test situation is the following:

- An appropriate test data generator is selected for the given test situation from the collection of test data generators.
- Input values involved in the computation are requested from the microprocessor model.
- The constraint formulated by the test situation is solved by the constraint solver engine on a basis of input values.

- Output values produced by the solver engine are assigned to instruction arguments (if they represent immediate values) or passed to the generator of initializing code (if they are stored in registers or memory).
- Initializing code is created by applying corresponding generators of initialization calls to the specified memory resources and input values.

The step that needs to be discussed in more detail is generation of initialization code. Different memory resources require different initialization code. Depending of the ISA, they may require a single call or a sequence of calls. The logic of initialization code generation is encapsulated in generators of initializing codes, which are identified by addressing modes that specify the destination for input values. Consequently, for each supported addressing mode a corresponding generator should be provided. When MicroTESK is creating initialization code for some instruction argument, it searches for an initialization call generator that uses the same addressing mode. Generators of initializing codes require configuration information. This information will be derived from Sim-nML specification enriched with specialized constructs. In the current prototype, generators of initializing calls have to be written by hand using API classes from the modeling library.

8.5 Constraint solver engine

To generate test data basing on constraints, the framework provides a constraint solver engine. It is based on Java Constraint Solver API [20], which was initially developed as part of MicroTESK, but now it is a separate project which is used in several other projects dedicated for hardware verification. To put it in a nutshell, the engine represents a collection of solvers oriented on specific tasks encapsulated behind a generic interface. This allows working with differed kinds of solvers in a uniform way. Solvers are divided into two major families: (1) standard general purpose solvers and (2) custom solvers aimed at specific tasks.

Standard solvers use some SMT solver implementation provided by a third-party vendor (for example, Z3 by Microsoft Research [13]). SMT solvers allow describing constraints as a set of assertions that should be hold for the specified variables. Assertions can be formulated for boolean expressions, arithmetic expression and expressions based on fixed-size bit-vectors. STM constraints can be used to limit the scope of random generation or to formulate preconditions for test situations. The framework provides wrappers implementing generic interfaces for all SMT solver features used in test data generation. This allows migrating to other SMT solver implementations without having to modify other parts of the testing framework. In the current implementation, interaction with solvers is performed via source files in the SMT LIB language [14]. The solver engine generates source files and passes them to an external SMT solver. The output data returned by the solver is parsed and packed into library classes. Here is an example of a constraint that describes an integer overflow situation using the SMT LIB language:

```
(define-sort Int_t () (_ BitVec 64))

(define-fun INT_ZERO () Int_t (_ bv0 64))
(define-fun INT_BASE_SIZE () Int_t (_ bv32 64))
(define-fun INT_SIGN_MASK () Int_t (bvshl (bvnot INT_ZERO) INT_BASE_SIZE))

(define-fun IsValidPos ((x!1 Int_t)) Bool
     (ite (= (bvand x!1 INT_SIGN_MASK) INT_ZERO) true false))
(define-fun IsValidNeg ((x!1 Int_t)) Bool
    (ite (= (bvand x!1 INT_SIGN_MASK) INT_SIGN_MASK) true false))
(define-fun IsValidSignedInt ((x!1 Int_t)) Bool
    (ite (or (IsValidPos x!1) (IsValidNeg x!1)) true false))

(declare-const rs Int_t) ; output variable
(declare-const rt Int_t) ; output variable

; rt and rs must contain valid sign-extended 32-bit values (bits 63..31 equal)
(assert (IsValidSignedInt rs))
(assert (IsValidSignedInt rt))

; the condition for an overflow: the sum is not a valid sign-extended 32-bit value
```

```
(assert (not (IsValidSignedInt (bvadd rs rt))))

; just in case: rs and rt are not equal (to make the results more interesting)
(assert (not (= rs rt)))

(check-sat) ; checks whether the constraint is satisfiable  (solves the constraint)
(get-value (rs rt)) ; gets values that lead to an overflow
```

Some testing tasks (e.g. covering cache or pipeline test situations) involve formulating constraints in terms of internal states of specific microprocessor model components. Functionality provided by SMT solvers is not suitable to solve such constraints. For these tasks special custom solvers can be provided. They are connected to the microprocessor model and use information on its internal state to produce test data that satisfy formulated conditions. They also can use SMT solvers to narrow the range of possible result values. When a coverage model is extended with new types of test situations, it often means a need to provide a corresponding custom solver. To facilitate extension of the engine with new solvers, both standard and custom solvers are implemented using a set of uniform interfaces generalizing services provided by these solvers.

To facilitate extension, the constraint solver engine hides solvers behind its public interfaces. In fact, the abstraction users of the engine operate with is constraints. Consequently, there are two categories of constraints: (1) standard and (2) custom. Both of them implement the same interfaces. However, their internal representations may be different as they require different solvers. The current version of the constraint solver engine supports only standard constraints and standard solvers. Constraints have the following attributes:

1. Name (a unique identifier).
2. Description (information that can be displayed to a user).
3. Solver identifier (specifies which solver should be used).

4. Variables. It can be input variables (in this case, they should be explicitly initialized) or output variables (in this case, they should be left uninitialized).

Standard constrains contain a hierarchy of objects that specify an SMT model represented by a set of assertions (or formulas) that must be satisfied. When a constraint is solved, a corresponding SMT solver checks the satisfiability of the model and suggests a solution (output variable values) that would satisfy that model. In an ideal case, to provide a better test coverage, each run of an SMT solver should return random values from the set of possible solutions. Unfortunately, the current implementation is limited to a single solution that is constant for all runs.

SMT models are described using context-independent syntax trees. Such a format is flexible as it is independent of a particular SMT solver implementation. To solve a constraint with a specific solver, the tree is traversed to produce input data in a format compatible with the given solver. The current implementation uses a limited set of SMT features. This makes the API compatible with a wide range of SMT solvers (most of free SMT solvers do not implement all SMT features). The syntax tree consists of nodes (Java objects) of the following types:

- Syntax. This is the root node of the tree. It holds the list of assertions (formulas) what specify conditions for the unknown variables associated with the constraint.
- Formula. Represents a single assertion expression. Can be combined with other formulas to build a more complex expression (by applying logic "or", "and" or "not" to it). The underlying expression must be a logic expression that can be solved to true or false.

- Operation. Represents an unary or binary operation with some unknown variable, some value or some expression as parameters.
- Variable. Represents an input variable. It can have an assigned value and, in such a case, will be treated as a value. Otherwise, it is an unknown variable. A variable includes a type as an attribute.
- Value. Specifies some known value of the specified type which can be accessed as an attribute.

The Operation, Variables and Value classes implement a common interface (the abstraction is called syntax element) and can be treated polymorphically. This allows combining them to build complex expressions. Figure 7 shows the UML diagram of classes that are used as modes of the syntax tree.

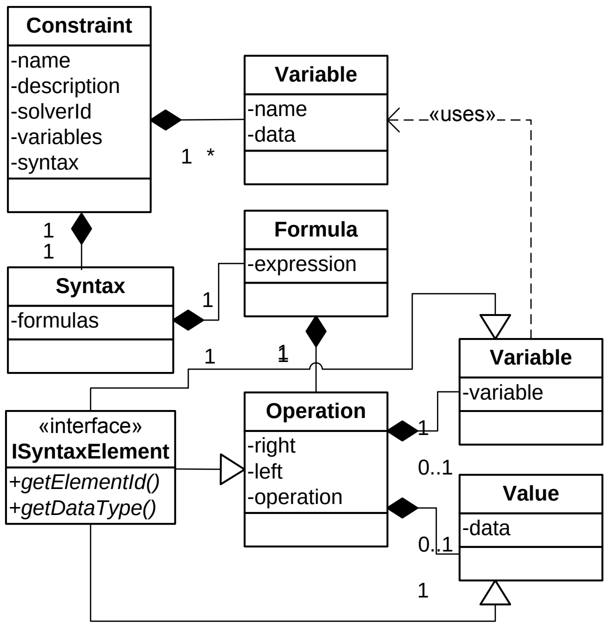

Figure 8. UML diagram of classes used in the SMT syntax tree

Constraint objects can be serialized to a hard disk. They are stored in the XML format. The format is flexible and extendable. The implementation supports adding new attributes to the stored objects and allows restoring objects from files that use different format versions. Below, there is an example of an XML file that describes a constraint. It demostrates the structure of the storage format. As it can be noticed, the hierarchy of XML tags closely resembles to the constraint syntax tree node hierarchy implemented in Java.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Constraint version="1.0">
  <Name>PowerOfTwo</Name>
  <Description>PowerOfTwo constraint</Description>
  <Solver id="Z3_TEXT"/>
  <Signature>
    <Variable length="32" name="x" type="BIT_VECTOR" value=""/>
  </Signature>
  <Syntax>
    <Formula>
      <Expression>
        <Operation id="BVUGT"/>
        <VariableRef name="x"/>
        <Value length="32" type="BIT_VECTOR" value="00000064"/>
      </Expression>
    </Formula>
    <Formula>
      <Expression>
        <Operation id="BVULT"/>
        <VariableRef name="x"/>
        <Value length="32" type="BIT_VECTOR" value="000000c8"/>
      </Expression>
    </Formula>
    <Formula>
      <Expression>
        <Operation id="EQ"/>
        <Expression>
          <Operation id="BVAND"/>
          <VariableRef name="x"/>
          <Expression>
            <Operation id="BVSUB"/>
            <VariableRef name="x"/>
            <Value length="32" type="BIT_VECTOR" value="00000001"/>
          </Expression>
        </Expression>
        <Value length="32" type="BIT_VECTOR" value="00000000"/>
      </Expression>
    </Formula>
  </Syntax>
</Constraint>
```

9. Trials: the ARM7TDMI architecture model

To conduct a trial of the implemented test program generation tool, a specification of a real (or, at least, close-to-real) microprocessor architecture is required. For this purpose, the ARM7TDMI [21], [22] microprocessor was chosen. ARM7TDMI (ARM7+Thumb+Debug+Multiplier+ICE) is a 32-bit RISC (Reduced Instruction Set Computer) microprocessor designed by ARM [24], [25] that implements the ARMv4T architecture [23]. ARM7TDMI was one of most widely used ARM cores in 2009 and its architecture is relatively simple. For this reason, it was selected to be used as an example of a design under test. An initial version of the Sim-nML specification of the ARM7TDMI microprocessor ISA was kindly provided by Indian Institute of Technology Kanpur [26]. The specification was subsequently modified to adapt it to the current implementation of MicroTESK (including error corrections, changes in structure, removing redundancies). It represents a full ARM7TDMI ISA specification excluding floating-point instructions that are not currently supported by MicroTESK and some other instructions that were considered redundant at the present stage. The table below contains the list of modeled ARM instructions:

Number	Name	Category	Description
1-2	B, BL	Branch instructions	Branch, and Branch with Link
3	BX	Branch instructions	Branch and Exchange Instruction
4	BLX	Branch instructions	Branch with Link and Exchange
5	MUL	Multiply instructions	Multiply
6	MLA	Multiply instructions	Multiply Accumulate
7	SMULL	Multiply instructions	Signed Multiply Long
8	UMULL	Multiply instructions	Multiply unsigned long
9	SMLAL	Multiply instructions	Signed Multiply Accumulate Long
10	UMLAL	Multiply instructions	Unsigned Multiply Accumulate Long

11	SMLA	Multiply instructions	Signed halfword Multiply Accumulate
12	SMLAL	Multiply instructions	Signed Multiply Accumulate Long
13	SMLAW	Multiply instructions	Signed halfword by word Multiply Accumulate
14	SMUL	Multiply instructions	Signed halfword Multiply
15	SMULW	Multiply instructions	Signed halfword by word Multiply
16-17	SWP, SWPB	Semaphore instructions	Swap and Swap Byte
18	MRS	Register access instructions	Move PSR to General-purpose Register
19	ADC	Data-processing instructions	Add with Carry (register and immediate)
20	ADD	Data-processing instructions	Add (register and immediate)
21	AND	Data-processing instructions	Logical AND (register and immediate)
22	BIC	Data-processing instructions	Bit Clear (register and immediate)
23	CMN	Data-processing instructions	Compare Negative (register and immediate)
24	CMP	Data-processing instructions	Compare (register and immediate)
25	EOR	Data-processing instructions	Exclusive OR (register and immediate)
26	MOV	Data-processing instructions	Move (register and immediate)
27	MVN	Data-processing instructions	Move NOT (register and immediate)
28	ORR	Data-processing instructions	Logical OR (register and immediate)
29	RSB	Data-processing instructions	Reverse Subtract (register and immediate)
30	RSC	Data-processing instructions	Reverse Subtract with Carry (register and immediate)
31	SUB	Data-processing instructions	Subtract (register and immediate)
32	SBC	Data-processing instructions	Subtract with Carry (register and immediate)
33	TST	Data-processing instructions	Test (register and immediate)
34	TEQ	Data-processing instructions	Test Equivalence (register and immediate)
35	LDR	Load and store instructions	Load Word

36	LDRB	Load and store instructions	Load Byte
37	LDRBT	Load and store instructions	Load Byte with User Mode Privilege
38	LDRT	Load and store instructions	Load Word with User Mode Privilege
39	STR	Load and store instructions	Store Word
40	STRB	Load and store instructions	Store Byte
41	CDP	Coprocessor instructions	Coprocessor Data Operations
42	CLZ	Miscellaneous arithmetic instructions	Count Leading Zeros
43	QADD	Parallel arithmetic instructions	Saturating Add
44	QDADD	Parallel arithmetic instructions	Saturating Double and Add
45	QSUB	Parallel arithmetic instructions	Saturating Subtract
46	QDSUB	Parallel arithmetic instructions	Saturating Double and Subtract
47	PLD	Preload data instruction	Preload Data

Table 3. Modeled ARM instructions

The total number of instructions described by the Sim-nML specification is 47 and the total size of the specification is about 3200 lines of code (LOC). This means that the average size of code required to describe a single instruction is about 68 LOC. Taking into account that the specification is raw and still contains redundancies (duplicating code that can be reused), it should be possible to decrease the average size of a single instruction specification to 55 LOC. The total size of Java source files of the generated model is about 16000 LOC. Consequently, the average size of a single instruction model is 340 LOC, which is approximately 5 times higher than the one of a Sim-nML specification.

Language	Total size (LOC)	Average size per an instruction (LOC)
Sim-nML	3200	68
Java	16000	340

Table 4. Sizes of the Sim-nML specification and Java model

Table 4 summarizes the discussed metrics. Based on the provided metrics, it can be concluded that automated generation of a microprocessor model from a formal specification requires 5 times less effort than manual development of a model in a high-level programming language (Java, in the given case). In fact, the metrics are incomplete since they cover only the design model and do not consider the coverage model, which has to be written by hand in both cases in the current version of the tool prototype. The size of the coverage model can be varied depending complexity of the architecture (number of possible states to be covered) and on applied knowledge extraction algorithms (the range of situations they are able to extract). However, the size of the coverage model code is expected to be at least 40% of the design model. This is means that the difference in the required effort between manual model development and automated model creation on a basis of formal specifications is expected to increase by at least 40% and is expected to amount at least 7 times.

The conducted trials demonstrated that the approach applied in MicroTESK helps significantly reduce the effort required to configure a test program generation tool for particular microprocessor architecture (to create a microprocessor model). This helps simplify development of functional tests and, consequently, help minimize time spent on functional verification of the microprocessor design.

10. Approbation and publications

The approaches to microprocessor verification described in the present work were also discussed in two publications [1] and [2]. The main ideas of the proposed solution and a prototype of the developed tool MicroTESK were presented at the following conferences and seminars:

1. 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2012), Perm, May 30-31, 2012.
2. 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2013), Kazan, May 30-31, 2013.
3. Design, Automation and Test in Europe (DATE 2013), University Booth exhibition, France, Grenoble, March 19-21, 2013
4. 50th Design Automation Conference (DAC 2013), the University Booth exhibition, USA, Texas, Austin, June 2-6, 2013
5. Seminar of the Software Engineering Department at Institute for System Programming of the Russian Academy of Sciences (ISPRAS), Moscow, April 16, 2013

11. Conclusion

In conclusion, it should be said that the overall goal of the project has been achieved. A microprocessor test program generation tool that uses formal specifications in Sim-mML as a source of information about microprocessor configuration and coverage goals has been designed and developed. The proposed approach helps minimize the effort and time required to configure the tool for a specific microprocessor architecture. This simplifies the process of functional verification and saves time of verification engineers. Also, flexible architecture of MicroTESK facilitates integration of new components into the tool (support for new DSL, test sequence generators, test data generators, constraint solvers, etc). This allows using the tools to solve a wide range of verification tasks that involve combining different techniques. Trials of the tool in which the ARM7TDMI microprocessor was used as an example of a design under test demonstrated advantages of the MicroTESK approach over manual development of a microprocessor model.

Due to time constraints, some features were implemented as prototypes that provide only a limited functionality. However, research and development will be continued in the future. The goal is to develop a fully-functional tool that can be used for solving real-life verification tasks in commercial projects. One the most important directions of future research is automation of extraction of coverage information from Sim-nML models.

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), pp. 64-69, Perm, Russia, May 30-31, 2012
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), Kazan, Russia, May 30-31, 2013
3. M. Freericks, The nML Machine Description Formalism. Techical Report,TU Berlin, FB20, Bericht 1991/15.
4. Surendra Kumar Vishnoi, Functional Simulation Using Sim-nML, master thesis, Indian Institute of Technology, Kanpur, 2006
5. 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, pp. 84-93, 2004
6. Information about RAVEN at ARM's web site, http://www.arm.com/community/partners/display_product/rw/ProductId/5171/
7. P. Mishra, A. Shrivastava and N. Dutt, Architecture Description Language (ADL)-Driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs, ACM Transactions on Design Automation of Electronic Systems, Vol. 11, No. 3, pp. 626–658, July 2006
8. P. Mishra and N. Dutt, Processor Description Languages: Applications and Methodologies, The Morgan Kaufmann Series in Systems on Silicon, 2008
9. MIPS64TM Architecture For Programmers. Revision 2.0. MIPS Tecnologies Inc., June 9, 2003
10. Terence Parr, "The Definitive ANTLR Reference: Building Domain-Specific Languages", The Pragmatic Bookshelf, 384 pages, 2007
11. Terence Parr, "Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages", The Pragmatic Bookshelf, 350 pages, 2009
12. Terence Parr, String Templates, http://www.stringtemplate.org/
13. L. Moura and N. Bjørner. Z3: An Efficient SMT Solver. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 337–340, 2008
14. D.R. Cok. The SMT-LIBv2 Language and Tools: A Tutorial. GrammaTech, Inc., Version 1.1, 2011
15. Ruby programming language, http://www.ruby-lang.org
16. JRuby, http://jruby.org/

17. A. Kamkin, E. Kornykhin and D. Vorobyev. s Reconfigurable Model-Based Test Program Generator for Microprocessors. Software Testing, Verification and Validation Workshops (ICSTW), 2011, pp. 47–54.
18. A. Kamkin. Test Program Generation for Microprocessors. Institute for System Programming of RAS, Volume 14, Part 2, 2008, pp. 23–63 (in Russian).
19. MicroTESK, http://forge.ispras.ru/projects/microtesk
20. Java Constraint Solver API, http://forge.ispras.ru/projects/solver-api
21. ARM7TDMI Technical Reference Manual, Revision: r4p1, ARM DDI 0210C, November26, 2004
22. http://en.wikipedia.org/wiki/ARM7TDMI
23. ARM Architecture Reference Manual, ARM DDI 0100I, ARM Limited, July 2005
24. http://en.wikipedia.org/wiki/ARM_architecture
25. http://arm.com/
26. Indian Institute of Technology Kanpur, Prof. Rajat Moona, http://www.cse.iitk.ac.in/users/moona/
27. M.S. Abadir, S. Dasgupta, Guest Editors' Introduction: Microprocessor Test and Verification. IEEE Design & Test of Computers, Volume 17, Issue 4, 2000, pp. 4–5.
28. A. Kamkin. Some Issues of Automation of Test Program Generation for Branch Units of Microprocessors. Institute for System Programming of RAS, Volume 18, 2010, pp. 129–150 (in Russian).