MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors

Alexander Kamkin
Software Engineering Department
Institute for System Programming of RAS
Moscow, Russian Federation
Email: kamkin@ispras.ru

Andrey Tatarnikov[1]
Software Engineering Department
Institute for System Programming of RAS,
National Research University Higher School of Economics
Moscow, Russian Federation
Email: andrewt@ispras.ru

Abstract— **Test program generation plays a major role in functional verification of microprocessors. Due to tremendous growth in complexity of modern designs and rigid constraints on time to market, it becomes an increasingly difficult task. In spite of powerful test program generators available in the market, development of functional tests is still known to be the bottleneck of the microprocessor design cycle. The common problem is that it takes significant effort to reconfigure a test program generation tool for a new microprocessor design. The model-based approach applied in the state-of-the-art tools, like Genesys-Pro, still does not provide enough flexibility since creating a microprocessor model is difficult and requires special knowledge and skills. The article suggests an approach to ease generator customization by using architecture specifications that describe the microprocessor behavior at a higher level. The approach is aimed at facilitating development of architecture models and, thus, minimizing time required to create functional tests. At the moment, we are working to implement a new generation of the test program generator MicroTESK that can be easily configured for various microprocessor architectures.**

Keywords—microprocessor design; architecture description languages; test program generation; functional verification; model-based testing.

I. INTRODUCTION

As modern microprocessors are becoming more and more complex, *functional verification* is becoming an increasingly difficult task. It is typical that up to half of resources spent on microprocessor design is devoted to verification. The most common approach to verification of microprocessors at a core level is *test program generation* (*TPG*) [1]. *Test programs* (*TPs*) are instruction sequences that trigger device events and optionally check validity of the resulting state of the microprocessor. A tool that creates test programs for a given microprocessor architecture in an automated way is usually referred to as a *test program generator* or a *TPG tool*. A well-known problem of TPG tools is that test generation logic is often tightly coupled with the architecture-specific knowledge, which makes the tool hard to maintain. In fact, a frequent solution to handle new microprocessor architecture is to rewrite the existing generator from scratch. As we can imagine, it increases cost of the microprocessor development and causes significant delays in the delivery schedule.

To make a TPG tool more flexible, the architecture-specific part has to be isolated from the test generation core. That is usually called *model-based TPG* [1]. Platform-dependent knowledge includes mainly an *instruction set model* (*ISM*) and *testing knowledge* (*TK*), a collection of design-specific *test situations* (conditions to be covered by tests). Typically, scenarios for microprocessor verification are described manually in the form of *test templates* (*TTs*). In abstracto, the idea of the method can be expressed by the formula ***TPs = TTs + TK + ISM*** (TPs are generated on the base of TTs, which are described in terms of the ISM and TK). The model-based TPG is a time-proved approach having been implemented in the industrial tools, like Genesys-Pro [1] and RAVEN [2]. However, creating a microprocessor's ISM and TK is rather difficult and requires special skills that verification engineers are usually lacking for.

In this article, we present a concept of *reconfigurable TPG* and its implementation in the MicroTESK tool. The key idea is to use *architecture description languages* (*ADLs*), which are commonly used in the area of functional simulation [3], for TPG configuration. *Architecture specification* (*AS*) in ADL is used by the tool to automatically build the microprocessor's ISM and TK. In addition to ASs, MicroTESK utilizes light-weight *configuration files* (*CFs*) for some microprocessor subsystems. This is due to the fact that some elements (e.g., cache memory, address translation mechanisms and others) are difficult to describe in general-purpose ADLs. Usage of high-level specifications and automated extraction of ISM and TK make it easy to adapt the tool for new architectures and to reconfigure it for several revisions of the same design. One more important feature of MicroTESK is its ability to automatically generate TTs of certain types. Thus, to generate TPs, one needs only AS and CFs (***TPs = AS + CFs***).

The MicroTESK generation core comprises tools and libraries that allow working with different configurations in a uniform way. To accomplish this, a flexible tool architecture has been proposed. It is based on a rather general microprocessor meta-model, which makes it possible to all architecture-dependent components (result of translation of the AS and CFs) to be accessed via architecture-independent APIs.

The rest of the paper contains an overview of the existing approaches to TPG and describes the MicroTESK principles and architecture.

[1]This work is partially supported by RFBR 11-07-12075-ofi-m.

II. RELATED WORK AND MOTIVATION

Hardware verification has always been a major issue for the research community. Over the last decades, a lot of hardware verification methods and tools have emerged. In fact, the idea of reconfigurable TPG is not new. It is based on a combination of well-known techniques. In this section, we will discuss the most significant of the existing approaches and industrial tools such as Genesys-Pro (IBM Research Lab) [1] and RAVEN (Obsidian Software Inc., now acquired by ARM) [2] implementing them.

Genesys-Pro is the best known TPG tool. It follows the model-based approach and operates with two kinds of knowledge: architectural model (ISM and TK) and TTs. To create an architecture model, some high level building blocks are provided. TK serves as a basis for creating TTs that describe verification scenarios. In TTs, it is possible to define constraints on individual scenario instructions (e.g., boundary conditions, exceptions, cache hits/misses, etc.). For each instruction of the TT the tool formulates a *constraint satisfaction problem* (*CSP*) and generates test data by solving the CSP. A known disadvantage of Genesys-Pro is that it is difficult to model instructions affecting memory devices [4]. Therefore, there are reasons to think that Genesys-Pro is hardly reconfigurable if significant modifications of the memory devices are required.

Another popular industrial solution is RAVEN. It is a tool that can generate fully random, semi-random or user-directed TPs for microprocessors. The tools components are separated into architectural models and TTs. Architectural models are developed by the tool vendor in collaboration with microprocessor manufacturers. For custom designs the *generator construction set* (*GCS*), a C++ API to the RAVEN core, is provided. There is no detailed information available on this technology. However, creating a model for RAVEN is unlikely to be an easy task for a verification engineer. In our opinion, it implies close interaction with the tool's developers, which is not convenient and will inevitably cause delays in the microprocessor verification process.

An interesting ADL-based approach to automated TPG is discussed in the work of Prabhat Mishra and Nikil Dutt [5]. It presents a concept of *graph-based functional test generation*. The approach uses the EXRESSION ADL to build a graph-based coverage model. The extracted model is automatically processed to extract test situations that will be covered by generated TPs. The test generation procedure is based on *model checking* (test is constructed as a *counterexample* for the negation of the target test situation). Heon-Mo Koo and Prabhat Mishra in their work [6] discuss a TPG technique that uses SAT-based *bounded model checking* (*BMC*) to generate TPs. Such an approach gives better results in terms of time and space required for counterexample generation compared to ordinary model checking that suffers from the state explosion.

Finally, it should be said that Institute for System Programming of RAS (ISPRAS) has already done some research and development on the TPG topic [4][7]. The present article summarizes the ideas that have been accumulated in ISPRAS and provides an overview of the research project our team is working on at the moment. The main motivation of the work is to propose a convenient way to describe microprocessor architectures that would reduce effort needed to create ISMs and TK.

III. MICROTESK OVERVIEW

MicroTESK is a reconfigurable TPG tool that uses ADL descriptions together with high-level configuration information to represent architecture-specific parts of the generator. By *reconfigurability* we mean an ability to easily switch to a new microprocessor design without having to modify the internal logic of the tool. General structure of MicroTESK is displayed in Figure 1. The tool uses two main types of input data: (1) design description and optionally (2) user-defined TTs. The former provides information about the target microprocessor, while the latter specifies scenarios to be reproduced in TPs. Outputs are TPs in an assembler language. MicroTESK functions can be divided into three major groups: (1) translating a design description into the ISM and extracting TK, (2) creating TTs on the base of the TK and (3) generating TPs from the TTs. In other words, to generate tests for the target microprocessor, we need to go through the following stages:

- Creation of a design description in ADL and configuration of the design subsystems. This task is performed by a *modeling engineer* who possesses knowledge about the microprocessor architecture.
- Translation of a design description and configuration into the architectural model (ISM and TK). This is done by a *translator* that uses unified building blocks from the *model library* to generate a model.
- Creation of TTs. TTs are created basing on the ISM and TK extracted from the design description. TTs can be either created automatically by a *TT generator* or provided by a *verification engineer*. The advantage of TTs is that they provide a flexible way to specify instruction sequences and instruction parameters that can vary depending on some conditions.
- Generation of TPs. TPs are generated by processing TTs. During this stage all CSPs formulated for test situations are solved and all instructions have their final parameter values assigned. In the end, a *TP generator* produces an assembler program which is compiled by a verification engineer into binary code and executed in a simulator or on a chip.

As we can notice, at each stage the tool works with data produced by the previous stage. This reduces dependencies between different components of the tool and facilitates their customization. For example, adding support for a new ADL will affect only the translator as the architectural model representation is independent of a particular ADL.

The next sections of the article describe the MicroTESK components in more detail.

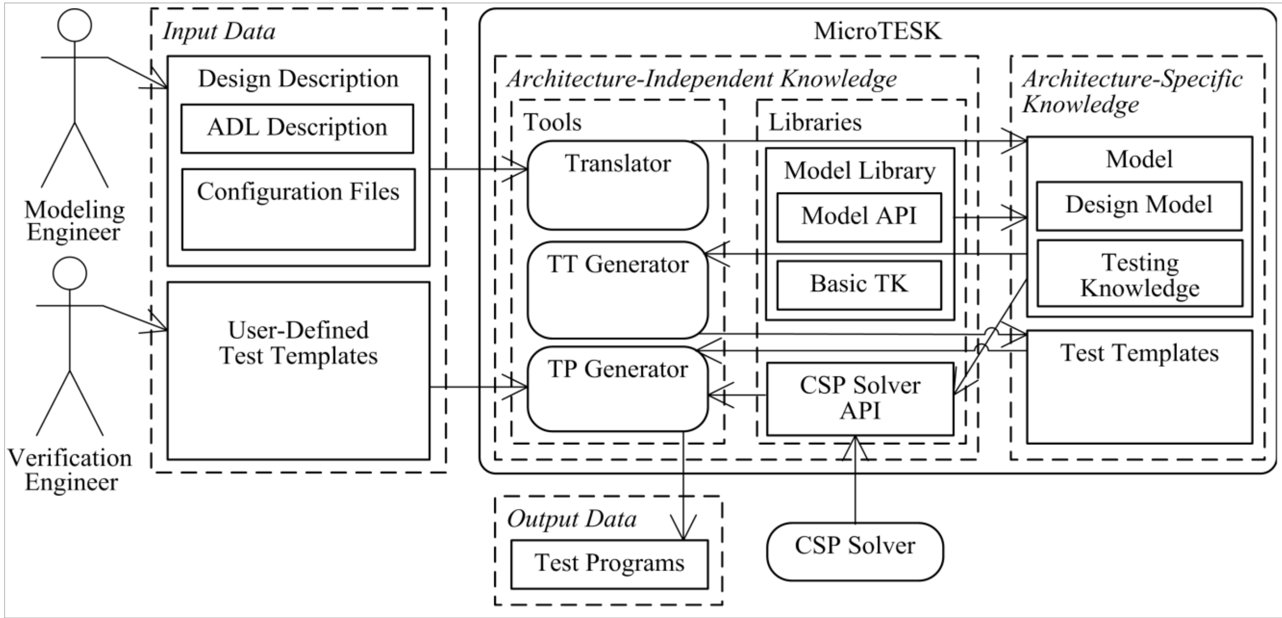

Figure 1. General structure of the MicroTESK TPG tool

IV. ARCHITECTURE MODELING

As it has already been said, MicroTESK makes use of ADLs to specify the target design architecture. At the moment, supported ADLs include nML [8] and Sim-nML [9]. nML is a formalism that describes a microprocessor at the instruction set level hiding unnecessary low-level details. The language is flexible and easy to use. Thereby, modeling a microprocessor architecture does not require significant effort. For example, a description of the integer addition instruction (ADD) from the MIPS instruction set architecture [10] looks as follows [4]:

```
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;
}

syntax = format("add %s, %s, %s",
  rd.syntax, rs.syntax, rt.syntax)

op ALU = ADD | SUB | …
```

As we can see, this notation is quite similar to the instruction's specification in the MIPS manual, which is shown below.

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 $temp_{32} \neq temp_{31}$ then
  SignalException(IntegerOverflow)
else
  GPR[rd] ← sign_extend($temp_{31..0}$)
endif

Basing on such specifications the microprocessor's TK can be automatically extracted. For example, analyzing the instruction description, we can derive three conditions that require attention:

1) The rt and rs general-purpose registers (GPRs) should contain sign-extended 32-bit values (bits 63..31 should be equal). Otherwise, the result of the instruction is UNPREDICTABLE. This means that under such a condition the microprocessor behavior is undefined and cannot be checked. Such situations should be avoided in TPs.

2) If the addition results in 32-bit two's complement arithmetic overflow, the destination register rd should not be modified and the IntegerOverflow exception should occur. Such a situation can be specified in a TT. When a TP is being generated the constraint solver engine will calculate exact values of the rt and rs GPRs to satisfy the constraint.

3) If the addition executes normally (does not cause an overflow), the sign-extended 32-bit result should be placed into the rd GPR. Such a condition can as well be used in TTs (for example, to be sure that some instruction does not raise exceptions).

ADL descriptions are used to build coverage models for individual instructions and to determine basic inter-instruction dependencies. It should be emphasized that models are composed from the building blocks provided by the model

library and independent of a particular ADL. Therefore, it is possible to use any ADL that provides enough information regarding the structure and behavior of microprocessors.

In addition to an instruction-level ADL description of the microprocessor, there is a need to specify some microprocessor subsystems, like memory management unit (MMU) and pipeline control unit (PCU), in more details. ADLs like Sim-nML are not suitable for describing these elements. At the same time, they should not be overlooked as it is very important to verify how a microprocessor handles events related to memory management and pipeline control. To provide specifications of these properties, special *configuration files* (*CFs*) are used.

MMU provides memory access protections, virtual-to-physical address translation and caching of instructions and data. It works with the main memory, cache memory (L1 and L2) and translation look-aside buffers (TLBs) that are used to accelerate virtual-to-physical address translation by caching latest translations. A cache or a TLB is represented by a memory buffer. At a logical level, each buffer is described as an array of sets of lines that can be specified as structures comprising several bit vectors called fields. A line stores a copy of memory data that has been recently read or written. Data is accessed by its address. When a buffer contains a line with a specified address the situation is called a hit; if it does not the situation is called a miss. When a miss occurs, the line is replaced with data stored in main memory at the given address. So, buffer configuration information includes the following attributes: set size (associativity), number of sets, line field description, address-to-index translation rule, rule for checking if a line and an address match and data displacement policy. To specify this information, we use CFs of the following kind [4]:

```
buffer L1 = {
  set = 4
  length = 128
  line = { tag:card(27), data:card(32) }
  index(addr:36) = { addr<8..2> }
  match(addr:36) = { addr<35..9> == tag<0..26> }
  policy = LRU
}
```

For the purpose of functional verification, there are two main situations that interest us: when a hit occurs and when a miss occurs. Both situations can be formulated as CSPs over the address being used to access data and state of the buffer [11][12].

Another important aspect related to architectural models is dependencies between instructions. Instructions change the state of the microprocessor and, thus, affect the behavior of subsequent instructions. For example, a precondition for the hit event is that corresponding data are loaded into cache, which is done by a previous instruction that accesses the same data line. To produce a complex instruction sequence that will give predictable results, we need first to simulate its execution to determine final parameter values of the dependent instructions. This is done at the final stage when MicroTESK processes TTs to produce TPs. The tool keeps a track of all events that occur in the model and provides this information to the TP generator Knowledge about possible dependencies between instructions is a part of TK extracted from the CFs.

V. CONSTRAINT SOLVER ENGINE

Important part of MicroTESK is a CSP solver engine. It facilitates generating test data and helps to achieve a better test coverage. Architecture specifications do not usually specify precise parameter values that lead to particular situations, but rather specify a class of possible values expressed as a set of conditions. For example, when we want to create a test for an integer overflow exception in the ADD instruction, we do not know values of parameter that cause the exception (in fact, there may be thousands of possible values). However, we know what conditions the resulting value should satisfy to recreate the situation. To generate parameter values that will make a test situation occur, the tool formulates a CSP and solves it with the help of the solver engine. The engine returns parameter values satisfying the constraint. Such an approach allows generating new test data from each time a TP is generated from the TT, which improves test coverage.

MicroTESK uses the SMT-LIB language [13] to formulate CSPs for test situations. CSP is expressed as a set of assertions that specify assumptions about values of input variables and results of operations performed with them. Modern solvers support bit vectors, which facilitates specifying constraints for data buffers used in different parts of microprocessor models (registers, cache, main memory, etc.). Below, there is an example of a CSP that specifies conditions leading to an integer overflow exception in the ADD instruction.

```
(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)
(declare-const rt Int_t)

; 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 summation
; result 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)

(echo "Values that lead to an overflow:")
(get-value (rs rt))
```

MicroTESK provides a possibility to generate CSPs automatically on the base of the TK extracted from the design description. It should be said that TK's constraints are stored in a format that is independent from a particular solver. The current version of the tool uses the Z3 solver by Microsoft Research [14]. The TP generator interacts with the solver via solver-independent *CSP solver API*. This allows the tool to use different solvers.

VI. TEST TEMPLATES

TTs are an important part of the MicroTESK solution. The tool provides facilities to create and modify TTs by hand. Despite the fact that some amount of TTs can be generated automatically based on TK, to cover all possible situations, it is often necessary to create TTs manually or customize automatically generated ones. Therefore, an expressive and easy-to-understand language is needed. Generally speaking, a TT describes a class of TPs that verify microprocessor behavior in particular test situations. Whereas TPs represent sequences of commands in a processor-specific assembler, TTs provide a way to describe a test scenario at a more abstract level. Such an approach gives a lot of advantages in terms of flexibility. For example, it allows generating tests taking into account dependencies between related instructions, create tests for a whole class of similar instructions and specify test parameters as ranges of possible values or as random values instead of hard-coding them. It also helps organize groups of separate test scenarios in more complex test cases and set up parallel test execution.

To describe TTs, a special test template description language (TTDL) is used. In the current version of the tool, it is based on the Ruby scripting language, which is extended with special automatically generated libraries that provide all hardware-related features and perform interaction with the design model. Generally, the TTDL features can be divided into the following groups according to their purpose:

A. Achitecture-related statements

Include constructs to simulate generalized processor-specific assembler instructions and a list of supported registers. Both instructions and registers can be combined into families. The TTDL allows specifying a family instead of a precise element or an address range instead of a specific address. Thus, it is possible to vary the level of randomness in the generated tests from completely random to completely directed. Also, we can specify dependencies between instructions. For example, we can make them use the same registers or the same address, which is selected at random when being accessed for the first time.

B. CSP-related statements

Constraints can be applied to instructions to recreate test situations. Typically, constraint conditions are extracted from ADL specifications. For example, it can be a condition that causes an integer overflow exception. Constraints are stored in a special catalog of constraints that includes information about instructions (or classes of instructions) they can be associated with. Constraints can be extracted from a design specification automatically, created manually or provided with the tools as independent general TK which is common for different microprocessors.

C. Generation flow statements

Provide control over instruction generation sequencing. Sometimes the sequence of instructions in a TP may need to be varied depending on some conditions or even to be randomized to achieve a better level of coverage. There are several possible ways to specify how a TP can be generated:

- As a sequence of ordered instructions (the order is specified in the TT);
- As a sequences of specified instructions given in a random order;
- As a sequence of instructions some of which are repeated depending on some conditions;
- As a sequence of instructions that contain instructions (or subsequences of instructions) randomly selected from the specified set of instructions;
- As a set of instruction sequences that should be executed concurrently;
- etc.

The TTDL language provides language constructs that offer such facilities. The set of offered features can be extended.

D. Standard language constructs

Include constructs derived from the underlying scripting language such as control flow operators, variables, constants and assertion statements. Such constructs are necessary to describe complex scenarios, to specify shared instruction parameters, to use common constants and to add validity checks to test scenarios.

E. Infrastructure-related statements

Provide a framework for creating TTs. Include base classes and global objects needed to organize the structure of TTs and provide communication with design models and CSP solvers during test generation. Some features are architecture-specific and are generated from models automatically.

The TTDL provides facilities that suffice for most verification tasks. To simplify the test design process, MicroTESK provides an ability to automatically generate some types of TTs. This includes templates for single instruction tests (that cover all possible execution paths for all supported instructions), combinatorial tests (that generate short sequences represented by combinations of specified instructions) and random tests (that produce random instruction sequences). Automated generation of TTs is performed based on TK extracted from the design description. To generate more specific TTs, the design model can be extended with additional information about test situations and instruction dependencies.

To illustrate the use of the TTDL, an example of a TT for a MIPS microprocessor is provided below.

```
class MyTemplate < Template
  def test()
    data = [ [0xEF, 0xFF], [0x1EF, 0x1FF], [0xFEF, 0xFFF] ];
    data.each { |d|
      xor r0, r0, r0;
      ori r(2), r0, d[0];
      ori r(4), r0, d[1];
```

```
    ld tmp1=r(1), 0x0, r(2);; hit([L1(), L2()], [25, 50, 75]);
    ld tmp2=r(3), 0x0, r(4);; hit([L1(), L2()], [25, 50, 75]);
    dadd r(5), tmp1, tmp2;; overflow;
  }
 end
end # class MyTemplate
```

This TT represents a scenario that generates a set of instruction sequences parameterized with data stored in the array. The generated sequences load data located at the addresses stored in the data array. For the ld instructions, the TT specifies constraints related to cache events:

```
hit([L1(), L2()], [25, 50, 75]);
```

This statement means that the line that accesses a memory device should cause a cache hit event to occur. It specifies a set of target caches and probabilities of the hit event occurrence. For this line, MicroTESK will generate a list of possible combinations and will add an instruction for each of them to the resulting TP. Another constraint is used to make the dadd instruction generate an overflow exception. The TTDL provides a wide range of facilities to express test situations that involve complicated series of events.

VII. CONCLUSION

Verification of modern microprocessors requires a lot of effort and efficient instruments. An ability to quickly reconfigure a TPG tool for a new design is a crucial requirement. In this paper, we offered a solution to the problem. The paper contributes the following approaches: (1) using high-level ADLs and CFs to specify the configuration of a target design and (2) building TK from high-level specifications basing on behavioral characteristics of the target design and (3) automated generation of TTs and TPs based on TK. The approaches are applied in MicroTESK, the instrument our team is working on. It makes use of the nML/Sim-nML ADL to describe target microprocessor designs. This formalism uses a format similar to the notation used in microprocessor manuals, which significantly facilitates creating configuration description of target devices. Another important application of ADLs is that they serve as source of behavioral characteristics of a microprocessor. MicroTESK is able to extract TK from ADL specifications and use it as a basis for creating test scenarios. This simplifies the job of a verification engineer who being armed with this knowledge can start creating tests as a soon as MicroTESK has processed an ADL specification. TTs are another major feature of MicroTESK. It provides a flexible way to specify complex test scenarios. Test situations can be formulated as CSPs, which eliminates the necessity to provide exact values of instruction parameters to make a particular event to occur.

The architecture of MicroTESK facilitates customization. Designs models are created based on an API provided by the model library. They are independent of a particular ADL and can be processed in a uniform way. Also, MicroTESK includes built-in TK about situations that are common for different microprocessors. The template generation logic combines built-in TK and TK extracted from the architecture model to generate test scenarios, which allows automating the process of creating tests for basic test situations. The tool can be extended to support new ADLs and new ways to describe TK and TTs. As we can see, MicroTESK automates most of activities required to create tests for a target microprocessor design, which helps significantly decrease delays in the delivery schedule.

At the present stage of our research, we implemented a prototype that supports only a small set of the described features. The first version of the prototype was tried with several industrial microprocessors and their subsystems. The experimental results are provided in the work of Kamkin, Kornykhin and Vorobyev [4]. Our current plans are to develop a full featured product that could be used by microprocessor vendors. A further direction of research is to more extensively automate creation of TTs. This will require using more complex models and test generation techniques. To keep in pace with temps of growth in complexity of modern microprocessor designs, TPG tools should provide more facilities to automate test design.

REFERENCES

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

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

[3] 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, July 2006, Pages 626–658.

[4] A. Kamkin, E. Kornykhin and D. Vorobyev, Reconfigurable Model-Based Test Program Generator for Microprocessors, A-MOST, Berlin, Germany, 2011.

[5] P. Mishra and N. Dutt, Graph-Based Functional Test Program Generation for Pipelined Processors, In Design Automation and Test in Europe (DATE), Paris, France, pages 182–187, February 16-20, 2004.

[6] H. Koo and P. Mishra, Functional Test Generation using SAT-based Bounded Model Checking, CISE Technical Report 05-008, Department of Computer and Information Science and Engineering, University of Florida, 2005.

[7] A. Kamkin. Test Program Generation for Microprocessors, Institute for System Programming of RAS, Volume 14, part 2, 2008, pp. 23–63 (in Russian).

[8] M. Freericks, The nML Machine Description Formalism, Techical Report, TU Berlin, FB20, Bericht 1991/15.

[9] R. Moona, Processor Models For Retargetable Tools, Proceedings of IEEE Rapid Systems Prototyping 2000 June 2000, pp 34–39.

[10] MIPS64™ Architecture For Programmers, Volume II: The MIPS64™ Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.

[11] E. Kornykhin, SMT-Based Test Program Generation for Cache-Memory Testing, East-West Design & Test Symposium (EWDTS), 2009, pp. 124–127.

[12] E. Kornykhin, Generation of Test Data for Verification of Caching Mechanisms and Address Translation in Microprocessors, Programming and Computing Software, Volume 36 Issue 1, 2010, pp. 28-35.

[13] D. R. Cok, The SMT-LIBv2 Language and Tools: A Tutorial, GrammaTech, Inc., Version 1.1, 2011.

[14] L. Moura and N. Bjørner, Z3: An Efficient SMT Solver, Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.