Contract Specification of Pipelined Designs: Application to Testbench Automation

Alexander Kamkin
Software Engineering Department
Institute for System Programming of Russian Academy of Sciences
25, B. Kommunisticheskaya, Moscow, 109004, Russia
E-mail: kamkin@ispras.ru

***Abstract*— In this paper we introduce a novel approach to formal specification of pipelined designs. The approach is based on contract specifications of pipe stages extended by a mechanism of temporal binding. Contract specifications describe the pipeline behavior in the form of preconditions and postconditions of pipe stages, while temporal binding combines specifications of separated stages into a co-operative specification. Such specifications are suitable for automatic generation of test oracles which check design correctness. The approach was integrated into the CTESK test development tool from the UniTESK toolkit. The methodology was successfully applied to several modules of the industrial microprocessor.**

I. Introduction

Modern world can not be imagined without huge variety of electronic devices. Mobile phones, digital cameras and briefcase computers have become the integral parts of human life. Special devices manage household appliances, control airplanes and space satellites, regulate medical systems of life support. Practically all of such systems are based on digital semiconductor hardware.

To make sure that hardware works correctly, i.e. meets all functional requirements stipulated in documentation, functional testing is usually used [1]. Requirements on thoroughness of hardware design testing are very strong. This is not only connected with the fact that hardware underlies all computer systems including safety-critical ones. Powerful influence on requirements is also exerted by economic forces.

In contrast to software, where an error correction (not error consequencies) does not cost anything, a post-silicon error in hardware can cause reproduction of all erroneous devices. The well-known FDIV bug in Pentium microprocessor consisted in incorrect division of some floating-point numbers cost Intel about $475 millions [2]. Time constraints of testing are also very strong. It is crucially important to develop system in time while it is much in demand.

Currently, the high-level hardware description languages (HDLs) are widely used in electronic design. The HDLs greatly speed up a development process by automated translation of the register-transfer-level (RTL) description into the gate-level netlist. However they can not guarantee that system developed is faultless. Thus, a testbench development does remain an acute and very important task.

State-of-the-art complexity of hardware designs does not allow to develop suitable test suites manually in reasonable time. The need of automated testbench development technologies is widely recognized. Development of such technologies and supporting tools has separated to a special branch of electronic design automation (EDA) industry which is known as *testbench automation*.

The main task of testing is to verify correspondence between design under test (DUT) behavior and requirements. To have the ability to do it automatically requirements should be represented in machine-readable form. Such form of requirements representation is usually called *formal specifications* or *specifications* for short.

The article focuses on specification and testing of pipelined designs. Pipelining is the key implementation technique used to make fast designs [3]. Generally, pipelining is the way of system organization whereby multiple operations are overlapped in execution. Each step in the pipeline, called *pipe stage*, completes a part of an operation; different steps are completing different parts of different operations in parallel.

Pipelining yields a reduction in the average execution time per operation, but it also introduces additional problems and new sources of errors. For example, two different operations can access the same resource on the same clock cycle. Such errors are called *control logic bugs*. According to statistics given in [4], 93.5% of MIPS R4000PC/SC errata (revision 2.2) [5] are control logic bugs.

We propose an approach applicable to complex industrial pipelined designs. The approach is based on contract specifications of pipe stages extended by a mechanism of temporal binding. Contract specifications describe the pipeline behavior in the form of preconditions and postconditions of pipe stages, while temporal binding combines specifications of separated stages into a co-operative specification.

The rest of the paper is organized as follows. The second section contains base definitions. In the third section contract specification of pipeline is considered. The fourth section comprises a short review of the UniTESK technology and the CTESK test development tool. In the fifth section related work is outlined. The sixth section is a case study. Finally, the seventh section concludes the paper.

II. Base Definitions

The classical model of finite state machine (FSM) extended with context variables, parameters, predicates, and functions

defined over context variables and parameters is known as *extended automaton* or *extended finite state machine* (EFSM), if automaton has a finite set of states and a finite transition relation [6]. Henceforward we will use the EFSM abbreviation even if a set of states or a transition relation are infinite.

The EFSM paradigm is widely used in computer science to model different kinds of software and hardware systems, such as real-time control systems, telecommunication protocols, etc. The FSM underlying an EFSM is said to model the control flow of a system, while context variables, parameters, and functions reflect its data flows. In this paper we use EFSM to model pipelined designs.

Hereinafter we will unify context variables and parameters of EFSM by general term *variables*. In the following definitions we suppose that each EFSM variable v is associated with a set of possible values D_v, which is called *domain* of variable v. If V is a set of variables, then D_V denotes a set of possible valuations of variables from set V.

Definition 1: An *EFSM* is a 6-tuple $\langle S, V, I \cup O, X, Y, T \rangle$, where:

- S is a set of *states*;
- V is a set of *context variables*;
- $I \cup O$ is a set of *input* and *output parameters*;
- X is a set of *stimuli*;
  Each stimulus $x \in X$ is parameterized by input parameters $in_x \subseteq I$; let D_x be a set of in_x valuations.
- Y is a set of *reactions*;
  Each reaction $y \in Y$ is parameterized by output parameters $out_y \subseteq O$; let D_y be a set of out_y valuations.
- T is a *transition* relation.
  Two subsets of V are specified for each transition $t \in T$:
  - $use_t \subseteq V$
    is a set of context variables *used* by the transition;
  - $def_t \subseteq V$
    is a set of context variables *defined* by the transition.

  Transition t is a 7-tuple $(s_t, x_t, y_t, \gamma_t, \lambda_t, \delta_t, s'_t)$, where:
  - $s_t \in S$
    is an *initial state* of the transition;
  - $\gamma_t : D_x \times D_{use_t} \to \{true, false\}$
    is a *guard condition* of the transition;
  - $\lambda_t : D_x \times D_{use_t} \to D_y$
    is a *reaction function* of the transition;
  - $\delta_t : D_x \times D_{use_t} \to D_{def_t}$
    is a *context update function* of the transition;
  - $s'_t \in S$
    is a *final state* of the transition. ■

Definition 2: A context variables valuation $\nu \in D_V$ is called a *context* of EFSM. ■

Definition 3: A pair $(s, \nu) \in S \times D_V$ is called a *configuration* of EFSM. ■

Definition 4: A predicate $pre_{s,x}(p, \nu)$, which is defined by the formula $\bigwedge_{t \in \{t' \in T | s_{t'} = s \wedge x_{t'} = x\}} \gamma_t(p, \nu)$, is called a *precondition* of stimulus x in state s. ■

Definition 5: A pair $x(p)$, where $x \in X$ and $p \in D_x$, is called an *initialized stimulus* of EFSM. ■

Definition 6: A pair $y(r)$, where $y \in Y$ and $r \in D_y$ is called an *initialized reaction* of EFSM. ■

Definition 7: A triple $t(p, r)$, where $t \in T$, $p \in D_{x_t}$, and $r \in D_{y_t}$ is called an *initialized transition* of EFSM. ■

Definition 8: An initialized transition $t(p, r)$ is said to be *enabled* for configuration (s, ν), if $s_t = s$, $\gamma_t(p, \nu) = true$, and $\lambda_t(p, \nu) = r$. ■

Denote the set of initialized stimuli, the set of initialized reactions, and the set of initialized transitions by $\overline{X}$, $\overline{Y}$, and $\overline{T}$ respectively.

The EFSM operates as follows. It receives an initialized stimulus and computes the set of enabled initialized transitions. A single transition nondeterministically chosen from the computed set fires. Executing this transition EFSM produces initialized reaction, updates context, and moves from the initial state of the transition to the final state. The EFSM usually starts from a designated configuration, which is called the *initial configuration*. A pair of an EFSM and an initial configuration is called an *initialized* EFSM.

III. Contract Specification of Pipeline

Intuitively, pipeline of length L is the way of system organization where execution of an operation consists of L successive stages, and it is possible to feed a next operation when the first stage of the previous one is complete. Let us formalize this idea.

Definition 9: A pair $(\overline{x}, l) \in \overline{X} \times \{1, ..., L\}$ is called a *stimulus state*. ■

Definition 10: A finite set of stimuli states $\{(\overline{x}_i, l_i)\}_{i=1}^{n}$ is called a *control state*; the empty set of stimuli states is called the *initial control state*. ■

Denote the set of stimuli processing states and the set of control states by $\overline{X}_L$ and $P_L \stackrel{def}{=} 2^{\overline{X}_L}$ respectively.

Definition 11: A contract specification of pipeline of length L is 6-tuple $\langle V, \nu_0, I \cup O, X \cup \{\tau\}, Z \cup \{\epsilon\}, \rho \rangle$, where:

- V is a set of *context variables*;
- $\nu_0 \in D_V$ is an *initial context*;
- $I \cup O$ is a set of *input* and *output parameters*;
- $X \cup \{\tau\}$ is a set of *stimuli*;
  Each stimulus $x \in X \cup \{\tau\}$ is parameterized by input parameters $in_x \subseteq I$; let D_x be a set of in_x valuations.
  Besides the set of input parameters, the following attributes are specified for stimulus x:
  - $use_x \subseteq V$
    is a set of context variables *used* by the stimulus;
  - $\gamma_x : P_L \to \{true, false\}$
    is a *guard condition* of the stimulus, such that:

$$\begin{cases} \gamma_x(\oslash) = true, \\ \gamma_x(\pi_1) \Rightarrow \gamma_x(\pi_2), \text{ if } \pi_2 \subseteq \pi_1; \end{cases}$$

  - $pre_x : D_x \times D_{use_x} \to \{true, false\}$
    is a *precondition* of the stimulus.

  Set of stimuli includes special stimulus τ, which is called *clock stimulus*, such that:
  - $in_\tau = \oslash$;

- $use_\tau = \oslash$;
- $\gamma_\tau \equiv true$;
- $pre_\tau \equiv true$.

- $Z \cup \{\epsilon\}$ is a set of *stages*;
  Each stage z is parameterized by output parameters $out_z \subseteq O$; let D_z be a set of out_z valuations.
  Besides the set of output parameters, the following attributes are specified for stage z:
  - $use_z \subseteq V$
    is a set of context variables *used* by the stage;
  - $def_z \subseteq V$
    is a set of context variables *defined* by the stage;
  - $\gamma_z : P_L \to \{true, false\}$
    is a *guard condition* of the stage, such that:
    $$\begin{cases} \gamma_z(\oslash) = true, \\ \gamma_z(\pi_1) \Rightarrow \gamma_z(\pi_2), \text{ if } \pi_2 \subseteq \pi_1; \end{cases}$$
  - $post_z : D_{use_z} \times D_z \times D_{def_z} \to \{true, false\}$
    is a *postcondition* of the stage.

  Set of stages includes special stage ϵ, which is called *empty stage*, such that:
  - $out_\epsilon = \oslash$;
  - $use_\epsilon = \oslash$;
  - $def_\epsilon = \oslash$;
  - $\gamma_\epsilon \equiv true$;
  - $post_\epsilon \equiv true$.
- $\rho : X \cup \{\tau\} \to (Z \cup \{\epsilon\})^L$ is a *mapping of stimuli to stages*, such that:
  - $\rho(\tau) = (\epsilon, ..., \epsilon)$;
  - $\gamma_{\rho_l(x)}(\{(\overline{x}, l)\}) = true$, for all $(\overline{x}, l) \in \overline{X}_L$. ■

Definition 12: A pair $z(r)$, where $z \in Z$ and $r \in D_z$, is called an *initialized stage* of contract specification. ■

If $y \subseteq Z$, $out_y = \bigcup_{z \in y} out_z$, and $r \in D_{out_y}$, then $y(r) \stackrel{def}{=} \{z(r|_{out_z}) \mid z \in y\}$.

A. Interpretation of Contract Specification

Contract specification of pipeline can be interpreted as a special case EFSM. States of the EFSM are control states of the contract specification; reactions of the EFSM are sets of the pipe stages executing simultaneously. Consider some definitions to begin with.

Definition 13: A stimulus $x \in X \cup \{\tau\}$ is said to be *enabled* on stage $l \in \{1, ..., L\}$ in control state $\pi \in P_L$, if $\gamma_{\rho_l(x)}(\pi) = true$; otherwise, it is said to be *locked*. ■

Definition 14: A set of stimuli states $enabled(\pi) \stackrel{def}{=} \{(\overline{x}, l) \in \pi \mid \gamma_{\rho_l(x)}(\pi) = true\}$ is called a *set of enabled stimuli* in control state $\pi \in P_L$. ■

Definition 15: A set of stimuli states $locked(\pi) \stackrel{def}{=} \pi \setminus enabled(\pi)$ is called a *set of locked stimuli* in control state $\pi \in P_L$. ■

To interpret a contract specification of pipeline we use two special functions: a *pipeline shift operator* to calculate a next control state, and a *temporal binding operator* to calculate a set of executing stages.

Definition 16: The *pipeline shift operator* is a function
$$\circ : (\overline{X \cup \{\tau\}}) \times P_L \to P_L$$
that for each pair $(\overline{x}, \pi) \in (\overline{X \cup \{\tau\}}) \times P_L$ possesses the value $\overline{x} \circ \pi$ which is a union of the following sets:

- $locked(\pi)$;
- $\{(\overline{x}, l+1) \mid (\overline{x}, l) \in enabled(\pi) \wedge l < L\}$;
- $\{(\overline{x}, 1)\}$, if $x \neq \tau$. ■

Definition 17: The *temporal binding operator* is a function
$$\theta : P_L \to 2^Z$$
that for each $\pi \in P_L$ possesses the value $\theta(\pi) \stackrel{def}{=} \{\rho_l(x) \mid (\overline{x}, l) \in enabled(\pi)\} \setminus \{\epsilon\}$. ■

Definition 18: A set of stages $\{z_1, ..., z_n\}$ is said to be *conflict*, if at least one of the following conditions is satisfied for some $1 \leq i \neq j \leq n$:

- $out_{z_i} \cap out_{z_j} \neq \oslash$ — conflict of reactions;
- $use_{z_i} \cap def_{z_j} \neq \oslash$ — read/write conflict;
- $def_{z_i} \cap def_{z_j} \neq \oslash$ — write/write conflict.

Otherwise, it is said to be *consistent*. ■

Definition 19: A control state π is said to be *consistent*, if $\rho_{l_1}(x_1) \neq \rho_{l_2}(x_2)$, for all different $(\overline{x}_{l_i}, l_i) \in \pi$, such that $\rho_{l_i}(x_i) \neq \epsilon$, and $\theta(\pi)$ is consistent. ■

Definition 20: A stimulus x is said to be *conflict* for a set of stages $\{z_1, ..., z_n\}$, if $use_x \cap def_{z_i} \neq \oslash$ for some $1 \leq i \leq n$. ■

Definition 21: A stimulus x is said to be *conflict* for a control state $\pi \in P_L$, if x is conflict for $\theta(\pi)$. ■

Definition 22: A contract specification is said to be *consistent*, if for all $\overline{x} \in \overline{X \cup \{\tau\}}$ and $\pi \in P_L$ the following conditions are satisfied:

- if x is conflict for π, then $\gamma_x(\pi) = false$;
- if π is consistent and $\gamma_x(\pi) = true$, then $\overline{x} \circ \pi$ is consistent. ■

Let $\Sigma = \langle V, \nu_0, I \cup O, X \cup \{\tau\}, Z \cup \{\epsilon\}, \rho \rangle$ be a consistent contract specification of pipeline, then it can be interpreted as the EFSM $\Delta = \langle S, V, (\pi_0, \nu_0), I \cup O, X \cup \{\tau\}, Y \cup \{\varepsilon\}, T \rangle$ with clock stimulus τ and empty reaction ε, which is organized as follows:

- $S = P_L$:
  the states of Δ are the control states of Σ;
- $\pi_0 = \oslash$:
  the initial state is the initial control state;
- $Y = 2^Z$:
  the reactions of Δ are the sets of Σ stages;
- $\varepsilon = \oslash$:
  the empty reaction is the empty set of stages;
- For all $\pi \in S$ and $x(p) \in \overline{X \cup \{\tau\}}$, such that $\gamma_x(\pi) = true$, transition relation T contains all initialized transitions $t = (\pi, x, y, \gamma, \lambda, \delta, \pi')$ of the following kind:
  - $use_t = use_x \cup (\bigcup_{z \in \theta(\pi')} use_z)$;
  - $def_t = \bigcup_{z \in \theta(\pi')} def_z$;
  - $\pi' = x(p) \circ \pi$;
  - $y = \theta(\pi')$;
  - $\gamma(p, \nu) = pre_x(p, \nu|_{use_x})$;

– the value of reaction function $\lambda \in D(\bigcup_{z\in\theta(\pi')} out_z)$ and the value of context update function $\delta \in D_{def_t}$ for all $\nu \in D_V$, such that $pre_x(p, \nu|_{use_x}) = true$, satisfy the predicate:

$$\Phi_{\pi,x}(\nu,\lambda,\delta) = \bigwedge_{z\in\theta(\pi')} post_z(\nu|_{use_z}, \lambda|_{out_z}, \delta|_{def_t}),$$

which is called the *test oracle* of stimulus x in control state π.

B. Correspondence between Specification and Implementation

An important concept that is generally used in functional testing is a *correspondence* between specification and implementation. Consider the contract specification $\Sigma = \langle V^S, \nu_0^S, I^S \cup O^S, X^S \cup \{\tau^S\}, Z^S \cup \{\epsilon^S\}, \rho^S\rangle$ and the initialized EFSM $\Delta = \langle S^I, V^I, (s_0^I, \nu_0^I), I^I \cup O^I, X^I \cup \{\tau^I\}, Y^I \cup \{\varepsilon^I\}, T^I\rangle$.

Definition 23: A surjective function $\varphi_S^{\rightarrow} : P_L^S \to S^I$, such that $\varphi_S^{\rightarrow}(\pi_0^S) = s_0^I$, is called a *state correspondence function.* ■

Definition 24: A function $\varphi_X^{\rightarrow} : \overline{X^S \cup \{\tau^S\}} \to \overline{X^I \cup \{\tau^I\}}$, such that $\varphi_X^{\rightarrow}(\overline{x}^S) = \tau^I$, iff $\overline{x}^S = \tau^S$, is called a *stimulus correspondence function.* ■

Definition 25: A bijective function $\varphi_Y^{\leftarrow} : \overline{Y^I \cup \{\varepsilon^I\}} \to 2^{\overline{Z}^I}$, such that $\varphi_Y^{\leftarrow}(\overline{y}^I) = \oslash$, iff $\overline{y}^I = \varepsilon^I$, is called a *reaction correspondence function.* ■

Definition 26: A function $\varphi_V^{\leftarrow} : D_V^I \to D_V^S$, such that $\varphi_V^{\leftarrow}(\nu_0^I) = \nu_0^S$, is called a *context correspondence function.* ■

Definition 27: An initial EFSM Δ is said to be *corresponding* to a contract specification Σ for a given set of correspondence functions $\langle\varphi_S^{\rightarrow}, \varphi_X^{\rightarrow}, \varphi_Y^{\leftarrow}, \varphi_V^{\leftarrow}\rangle$, if for all $\pi^S \in P_L^S$, $x^S(p^S) \in \overline{X^S \cup \{\tau^S\}}$ and $\nu^I \in D_V^I$ if $\gamma_{x^S}(\pi^S) = true$, then $x^I \in init(s^I)$, and if $pre_{x^S}(p^S, \nu^S) = true$, then $pre_{s^I,x^I}(p^I, \nu^I) = true$, where $s^I = \varphi_S^{\rightarrow}(\pi^S)$, $x^I(p^I) = \varphi_X^{\rightarrow}(x^S(p^S))$, $\nu^S = \varphi_V^{\leftarrow}(\nu^I)$. At that rate for each initialized transition $t^I(p^I, r^I) \in init(s^I, x^I)$, which is enabled in configuration (s^I, ν^I), $s'_{t^I} = \varphi_S^{\rightarrow}(\pi'^S)$, $\varphi_Y^{\leftarrow}(y_{t^I}(r^I)) = \theta(\pi'^S)(r^S)$, and $\Phi_{\pi^S,x^S}(\nu^S, r^S, \nu'^S) = true$, where $\pi'^S = x^S(p^S) \circ \pi^S$ and $\nu'^S = \varphi_V^{\leftarrow}(\delta_{t^I}(\nu^I))$. ■

The definition above gives the following scheme for checking correspondence between specification and implementation during a testing:

```
(π^S, ν^S) ← (⊘, ν_0^S);
if(s^I ≠ φ_S^→(π^S) ∨ ν^S ≠ φ_V^←(ν^I)) Error();
while(¬ isTestComplete()) {
  x^S(p^S) ← getNextStimulus();
  if((γ_{x^S}(π^S) ∧ pre_{x^S}(p^S, ν^S)) {
    x^I(p^I) ← φ_X→(x^S(p^S));
    applyStimulus(x^I(p^I));
    y^I(r^I) ← waitForReaction();
    y^S(r^S) ← φ_Y^←(y^I(r^I));
    ν'^S ← φ_V^←(ν^I);
    π^S ← x^S(p^S) ∘ π^S;
    if(s^I ≠ φ_S^→(π^S)) Error();
    if(y^S ≠ θ(π^S)) Error();
    if(¬Φ_{π^S,x^S}(ν^S, r^S, ν'^S)) Error();
  }
}
```

Sufficiently often the implementation state s^I is hidden. Furthermore, the function of state correspondence $\varphi_S^{\rightarrow}$ is actually undefined. Thus, checks of kind $s^I = \varphi_S^{\rightarrow}(\pi^S)$ are usually omitted. The same is true for comparison $y^S \neq \theta(\pi^S)$, because one can consider reactions as the parts of the context. So, the only check that is generally done by testbench in each step of testing is $\Phi_{\pi^S,x^S}(\nu^S, r^S, \nu'^S) = true$.

IV. UniTESK Technology

The UniTESK technology [7], [8] was developed at the Institute for System Programming of Russian Academy of Sciences [9]. The original purpose of the technology is the development of high-quality functional tests for software systems. The UniTESK technology and supporting tools have been successfully applied for testing different kinds of software (operating systems, telecommunication protocols, real-time systems, etc.). A key moment in the successful use of the UniTESK technology is the flexible and scalable test system architecture, which allows to adapt the technology to various classes of systems [8].

The main idea of UniTESK is separating the test sequence generation from the DUT behavior specification. On the one hand, it uses FSM traversal techniques to generate test sequences; on the other hand, it utilizes formal specifications in the form of preconditions and postconditions of operations to describe the DUT behavior. The approach has distinctive feature that it does not operate with explicit FSM models. In contrast, it works with implicit ones being described by means of FSM state calculation function and set of possible stimuli for each of the reachable states.

A. UniTESK Test System Architecture

UniTESK test system architecture has been developed as a result of many years experiments on specification-based testing of the industrial software from different fields and of different levels of complexity [8]. These experiments have allowed to create the flexible and scalable test system architecture based on the following division of the testing task into subtasks:

- Generation of a test sequence to achieve the functional coverage needed;
- Creation of a single stimulus within the test sequence;
- Building a connection between the test system and DUT;
- Verification of the DUT behavior in response to a single stimulus.

Special test system components are provided by UniTESK technology to solve these subtasks. Their interactions are shown on **Fig. 1**.

Test engine is a library component of the UniTESK test system. Test engine and test action iterator are intended for test sequence generation. Test engine is based on an FSM traversal algorithm.

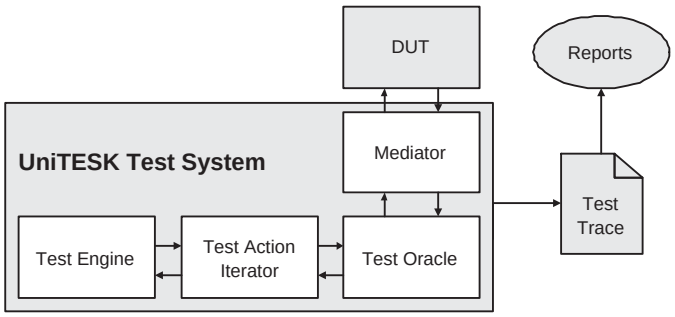

Fig. 1. UniTESK test system architecture.

Test action iterator works under the test engine control. It calculates current FSM state, iterates corresponding stimuli, and applies them. Test action iterator is automatically generated from the high-level test scenario description.

Test oracle verifies the DUT behavior in response to a single stimulus. It is automatically generated from the formal specifications.

Mediator connects formal specifications and DUT implementation. It makes some transformations of the stimuli and reactions and also synchronizes the specification state with the implementation one.

Test trace shows the events happening during the testing. It is used by the UniTESK supporting tools to automatically generate different reports that help in the test results analysis.

B. CTESK Test Development Tool

CTESK test development tool is an implementation of the UniTESK conception for the C programming language. It uses SeC (specification extention of C) language to develop test system components. SeC language provides test developers with special functions:

- *Specification functions* – to specify DUT operations and to define functional coverage structure;
- *Mediator functions* – to connect specification functions with corresponding stimuli;
- *Function of FSM state calculation* – to calculate FSM state on the base of the specification state;
- *Scenario functions* – to define a set of stimuli to be applied in each of the reachable states.

CTESK tool has been used for testing Verilog HDL and SystemC designs of hardware. The detailed information on this subject is presented in [10] and [11].

C. Specification of Pipelined Designs

We have adapted the CTESK tool for contract specification of pipeline designs. To illustrate basic ideas of the suggested approach let us consider an example of a 3-stages floating-point adder. The adder is intended for adding two normalized single-precision floating-point numbers (zero values are also permitted) [12]. The pipeline of the unit consists of 3 stages: (1) alignment of exponents, (2) addition of fractions, and (3) normalization of result.

The first step of operation specification is the definition of so-called *operation descriptor type* that describes the current state of operation execution. This type usually contains operands of the operation and all kinds of temporal values calculated on one stage to be used on successive stages.

```
// Descriptor of ADD operation
specification typedef struct ADDDescriptorT {
  // Operation operands
  bool op1_sign;
  uint8_t op1_exponent;
  uint32_t op1_fraction;
  ...
} ADDDescriptorT;
```

The specification function of the operation contains operation precondition.

```
// Specification function of ADD operation
specification void ADD_spec(SingleT op1, SingleT op2) {
  // Operation precondition
  pre {
    return (isZero_SingleT(op1) || isNormalized_SingleT(op1))
      && (isZero_SingleT(op2) || isNormalized_SingleT(op2));
  }
  ...
}
```

For each stage of the operation a special function is developed to set forth stage requirements. Let us consider specification of the following requirement: ”If operands have different exponents, then fraction of the operand with the smaller exponent is shifted to the right (the number of positions that the bits in the fraction are to be shifted is the difference between exponents). If there is a unit among the shifted bits, then output *inexact_align* is set to high; otherwise, it is set to low.”

```
// Specification of the ALIGNMENT stage
reaction ADDDescriptorT* ADD_align_spec(void) {
  AdderUnitT *adder_unit = getAdderUnit();
  ADDDescriptorT *add = ADD_align_spec;
  post {
    if(add->op1_exponent > add->op2_exponent) {
      int shift = add->op1_exponent - add->op2_exponent;
      return adder_unit->inexact_align ==
        (add->op2_fraction & mask(shift)) != 0;
    }
    if(add->op1_exponent < add->op2_exponent) {
      int shift = add->op2_exponent - add->op1_exponent;
      return adder_unit->inexact_align ==
        (add->op1_fraction & mask(shift)) != 0;
    }
    ...
  }
}
```

In the code above *adder_unit* denotes a variable that stores specification representation of the current DUT state.

V. Related Work

Functional testing of pipelined designs is mostly researched in the area of microprocessor design. The good many articles are dedicated to the methods of test generation for pipelined microprocessors. Many researchers come to a consensus that specification-based testing is the right direction for functional testing of hardware designs. The main question is which models and notations should be used.

The important step toward the specification-based testing of pipelined microprocessors was made by Ho et al. [4]. They suggest deriving pipeline control logic as FSM, analyzing reachable states of the FSM, and traversing the FSM to produce test vectors. Unfortunately, this methodology is not applicable to complex industrial designs [13].

Existing methods of test generation utilize explicit cycle-accurate specifications, e.g. Mishra et al. [14], [15], [16], [17] and Ur et al. [13] use explicit SMV specifications to describe DUT behavior and to generate test programs. Two modern directions of test generation are:

- test generation using FSM traversal techniques (Ho et al. [4], Ur et al. [13]);
- test generation using model checking techniques (Mishra et al. [14], [15], [16], [17]).

We suppose that model checking does not scale well on complex industrial designs. However some techniques, like bounded model checking (BMC) and properties decomposition, allow to achieve suitable efficiency of test generation [18], [19].

We use FSM models to generate test sequences, but in contrast to other methods we use implicit models being described by means of FSM state calculation function and set of possible stimuli for each of the reachable states. Transition relation of the FSM is built on-the-fly during the testing. We suppose that the use of the implicit models for specification and testing increases the scalability of the approach.

VI. Case Study

The suggested approach was applied to RTL model of translation lookaside buffer (TLB) of the industrial microprocessor with MIPS64^{TM}-compatible architecture [20], [21]. TLB is a buffer in a microprocessor that is used to increase the speed of address translation. TLB has a fixed number of entries containing part of the page table which translates virtual addresses into physical ones.

The memory of the TLB under test comprises three buffers: a 4-entries instruction micro TLB (ITLB), a 4-entries data micro TLB (DTLB), and a large 64-entries joint TLB (JTLB). The purpose of the micro TLBs is to allow two address translations to be performed simultaneously – one for an instruction fetch address (via the ITLB) and one for a data load/store address (via the DTLB).

TABLE I
TLB requirements partitioning

Operation	Multistage	Requirements
Read	No	7
Write	No	9
Probe	No	8
Data Address Translation	Yes	38
Instruction Address Translation	Yes	37
Total	—	99

If a translation is not found in the corresponding micro TLB, then the JTLB is accessed. Once the translation is retrieved, it is written back to the micro TLB. To refill micro TLBs the least-recently-used (LRU) algorithm is used – micro TLBs always replace the entry which has not been accessed for the longest amount of time. Thus, micro TLBs contain a subset of translations that are most-recently-used.

Address translation operations are organized as multistage pipelined operations. A micro TLB miss sequence has a penalty of one extra clock cycle. If we have simultaneous ITLB miss and DTLB miss, the DTLB gets first priority when accessing the JTLB, and the translation of instruction address stalls an additional cycle, giving a total penalty of two latency cycles.

Besides the address translation operations, the TLB under test implements operations for reading entry from the buffer, writing entry to the buffer, and probing if the entry exists in the buffer. The interface of the TLB under test contains about 30 inputs and as many outputs. The RTL model of the TLB is implemented in Verilog HDL. The source code of the model makes up to 8 KLOC.

The process of testbench development was organized as follows. Part of the requirements on the TLB under test were formulated by the developers, while the others were derived from the technical documentation. A total number of the requirements is about a hundred. TLB requirements partitioning is shown in **Table 1**.

Requirements on each operation were represented in the form of preconditions and postconditions of pipe stages. After that, pipeline shift operator and temporal binding operators were defined. It should be emphasized that all requirements were cheaply formalized. The volume of specifications makes up to 2.5 KLOC in SeC language.

We have found 9 errors in the TLB implementation including critical ones. Errors statistics is shown in **Table 2**. It should be noted that all errors were found in the multistage operations and the majority of errors (67%) are connected with the control logic. The total labor costs of the testbench development make up to about 2.5 man-months.

TABLE II

TLB ERRORS STATISTICS

Type of Errors	Number	Percent
Datapath Errors	3	33%
Control Logic Errors (Incorrect State)	5	56%
Control Logic Errors (Data Hazards)	1	11%
Total	9	100%

VII. CONCLUSION

The need of automated testbench development for complex pipelined designs is widely recognized. The paper described the novel approach to formal specification of pipelined designs, which is suitable for testbench automation. The approach is based on contract specifications of pipe stages extended by mechanism of temporal binding. Such specifications are suitable for automatic generation of test oracles which check DUT correctness. The approach was integrated into the CTESK test development tool from the UniTESK toolkit. The methodology was successfully applied to several modules of the industrial microprocessor.

The roots of the approach are Design-by-Contract (DbC) [22] and UniTESK [7]. As Ur et al. [23], we believe that close integration between the software testing and hardware validation communities can benefit both of them. Actually, many useful approaches that are developed and proven to be successful in one domain are rarely used in the other. Of course, this does not mean that all problems of one community can be solved by the other, but general problems are the same.

REFERENCES

[1] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[2] B. Beizer. *The Pentium Bug – An Industry Watershed*. Testing Techniques Newsletter, TTN Online Edition, September 1995.

[3] D. Patterson and J. Henessy. *Computer Organization and Design*. 3^{rd} Edition, Morgan Kaufmann, 2005.

[4] R. Ho, C. Yang, M. Horowitz, and D. Dill. *Architecture Validation for Processors*. ISCA'95: International Symposium on Computer Architecture, 1995.

[5] *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*. MIPS Technologies Inc., May 10, 1994.

[6] A. Petrenko, S. Boroday, and R. Groz. *Confirming Configurations in EFSM Testing*. IEEE Transactions on Software Engineering, 2004.

[7] http://www.unitesk.com

[8] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. *UniTESK Test Suite Architecture*. FME'02: Formal Methods Europe. LNCS 2391, Springer-Verlag, 2002.

[9] http://www.ispras.ru

[10] V. Ivannikov, A. Kamkin, V. Kuliamin, and A. Petrenko. *Application of the UniTESK Technology to Functional Testing of Hardware Designs*. Preprint 8, Institute for System Programming of Russian Academy of Sciences, Moscow, 2005 (in Russian).

[11] A. Kamkin. *The UniTESK Approach to Specification-Based Validation of Hardware Designs*. ISoLA'06: The 2^{nd} International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, November 2006.

[12] IEEE 754-1985. *IEEE Standard for Binary Floating-Point Arithmetic*. NY: IEEE, 1985.

[13] S. Ur and Y. Yadin. *Micro Architecture Coverage Directed Generation of Test Programs*. DAC'99: Design and Automation Conference, 1999.

[14] P. Mishra and N. Dutt. *Automatic Functional Test Program Generation for Pipelined Processors using Model Checking*. HLDVT'02: The 7^{th} IEEE International High-Level Design Validation and Test Workshop, 2002.

[15] P. Mishra and N. Dutt. *Architecture Description Language Driven Functional Test Program Generation for Microprocessors using SMV*. CECS Technical Report 02-26, September 13, 2002.

[16] P. Mishra and N. Dutt. *Graph-Based Functional Test Program Generation for Pipelined Processors*. DATE'04: Design, Automation and Test in Europe Conference and Exhibition, 2004.

[17] P. Mishra and N. Dutt. *Functional Coverage Driven Test Generation for Validation of Pipelined Processors*. DATE'05: Design, Automation and Test in Europe, 2005.

[18] H.M. Koo and P. Mishra. *Test Generation using SAT-based Bounded Model Checking for Validation of Pipelined Processors*. ACM Great Lakes Symposium on VLSI, 2006.

[19] H.M. Koo and P. Mishra. *Functional Test Generation using Property Decomposition for Validation of Pipelined Processors*. DATE'06: Design, Automation and Test in Europe, March 2006.

[20] http://www.mips.com/content/Products/Architecture/MIPS64

[21] *MIPS64 Architecture For Programmers*. Revision 2.0. MIPS Tecnologies Inc., June 9, 2003.

[22] B. Meyer. *Applying 'Design by Contract'*. IEEE Computer, vol. 25, #10, October 1992.

[23] S. Ur and A. Ziv. *Cross-Fertilization between Hardware Verification and Software Testing*. SEA'02: Software Engineering and Applications, ACTA Press, 2002.