A TLM-based approach to functional verification of hardware components at different abstraction levels

Mikhail Chupilko
Institute for System Programming of RAS
A. Solzhenitsyn st. 25, 109004
Moscow, Russia
chupilko@ispras.ru

Alexander Kamkin
Institute for System Programming of RAS
A. Solzhenitsyn st. 25, 109004
Moscow, Russia
kamkin@ispras.ru

Abstract—**Verification has long been recognized as an integral part of the hardware design process. When designing a system, engineers usually use various design representations and concretize them step by step up to a physical layout. At the beginning of the process, when there is much of indeterminacy, only abstract reference models are applicable to verification; when the process is close to the end, more concrete ones can be utilized. The article concerns problems of developing reusable verification systems (testbenches), which can be used to analyze different versions of the same component at different abstraction levels. We suggest an approach to construct reusable reaction checkers basing on a concept of Transaction Level Modeling (TLM). The paper includes general description of the approach, considers several particular cases, and outlines our experience.**

Keywords-hardware design; functional verification; simulation-based-verification; co-simulation; testbench automation; reaction checking; transaction level modeling (TLM)

I. INTRODUCTION

Nowadays, hardware systems become more and more complex consuming lots of resources for their design and test. Two main approaches to overcome the complexity are *decomposition* and *abstraction*. In general terms, decomposition is partitioning of a system into a set of components, while abstraction is extraction of essential properties of the components with respect to some aspect of interest. Hardware designing can be considered as an evolutionary process of system decomposition and concretization driven by the target requirements and available resources.

To check a design's compliance with the specification, *functional verification* is used. Many engineers consider verification to be one of the final stages of the design process. However, that is not true. Verification runs through the entire process providing on-time feedback and making the process more controllable. Due to the designing nature, verification has to deal with individual components as well as with the whole system. What is also important is that verification should use different kinds of models at different design stages – more *abstract* ones at the beginning and more *concrete* ones at the end.

The paper focuses on verification of individual components of hardware systems. It is assumed that a system is divided into a set of components and the general functionality of the components does not change. The problem we are going to solve is to develop a *verification system* (so-called *testbench* [1]) for a given component so that it could be reused during the component's design cycle. The most usable way to develop an automated testbench is based on *co-simulation*, when along with the target component an independent *reference model* is created and used for checking the component's behavior (its observable *reactions*). Since a reference model is a core part of a testbench, our research is concentrated on organization of such models and their adaptation to the co-simulation process.

We propose a methodology for creating reusable *reaction checkers* (including reference models as a part) that is based on *Transaction Level Modeling* (*TLM*) [2]. The key feature of TLM is separation of communication from computation. System communication is modeled by *channels*, while *transactions* (in other words, data transmission) take place by calling interface functions of those channel models. We suggest a TLM-based reaction checker's architecture suitable for all abstraction levels (thus applicable to all design stages). Reuse of a reaction checker is attained by refining or changing the architecture's components. The architecture is proved to be flexible and universal in several real-life industrial projects.

The article is organized as follows. Section 2 reviews the related work. Section 3 introduces TLM and identifies the main abstraction levels being used in hardware design. Section 4 describes the suggested approach to testbench organization. Section 5 considers application of the approach at different abstraction levels. Section 6 gives a classification of errors in hardware basing on the suggested reaction checking scheme. Section 7 outlines our experience. Section 8 concludes the paper.

II. RELATED WORK

Two main approaches to functional verification of hardware designs are *formal methods* and *simulation-based methods* [3]. It is well known that formal methods are exhaustive (in a sense) but do not scale well and can be applied only at the later design stages, when the requirements are rather stable. Simulation-based methods are not exhaustive, but they are much more flexible and thereby employed at different stages.

978-1-4577-1490-0/11/$26.00 ©2011 IEEE

A testbench is an environment used to verify the design correctness via simulation. A typical testbench has three components: (1) a *stimulus generator*, (2) a *reaction checker* (*test oracle*), and (3) a *coverage tracker*. The stimulus generator creates input stimuli to the design under verification (DUV). The reaction checker estimates the correctness of the design's behavior. The coverage tracker evaluates the testing completeness. Let us consider the existing approaches to reaction checking. Nowadays, there are three basic methods: (1) *self-checking tests*, (2) *assertions*, and (3) *co-simulation*.

Self-checking tests are an old-age approach to testbench organization in which each stimulus (to be more precise, each test case) is encoded with a procedure of checking for the expected results [4]. In this approach, each test case should perform reaction checking during and at the end of the test. This has certain disadvantages. First, it is really difficult to write test cases checking sophisticated test sequences. Second, test cases require maintenance during the design process to keep up with the design's changes. Finally, the self-checking approach suffers from incompleteness, because each test case checks only few aspects of the design's behavior.

Assertions are statements about a design's intended behavior, which must be verified [5]. In this method, the checks are detached from the stimuli and injected into the design's code (or written in separate files). There is no need for hand-written test cases checking for the specific results – instead, automatic test generation can be used for verification. Assertions usually state the most critical or the most obvious DUV's properties. Therefore, assertion-based checking usually lacks for completeness. It should also be said that in case of built-in assertions it is impossible to create a reaction checker until the design is fully described.

Co-simulation is a method for reaction checking in which an independent reference model is used along with the target design model [4]. The two models are co-simulated using the same test sequences and their results are compared for equality. Every mismatch is tracked down to discover which of the models is incorrect. The usage of a reference model allows generating tests automatically. However, making two models agree for all test sequences is a difficult task, which in some cases is almost tantamount to describing two equivalent designs.

Let us analyze the methods described above. Self-checking tests do not provide high-level automation and suffer from incompleteness of reaction checking. Furthermore, they are hard to maintain during the design process. Assertions are a perfect solution for checking a few numbers of properties but are not suitable for gap-free checking of complex hardware components. Co-simulation seems to be the most promising approach, but it requires development of a reference model, which is labor intensive. To simplify development and maintenance of reference models, a special methodology is surely needed. In our opinion, TLM is a good basis for such kind of methodology.

These thoughts underlie Open Verification Methodology (OVM), a well-known verification approach [6]. According to OVM, a testbench should be subdivided into several functional components and interfaces between them should be specified by means of TLM. The methodology describes the basic testbench architecture but says nothing about how the architecture's components are organized. For example, it does not answer how to generate complex test sequences and how to check the design's behavior.

III. TLM AND DIFFERENT ABSTRACTION LEVELS

TLM is an approach to model hardware systems where details of communication between components are separated from computation [2]. Communication mechanisms such as busses and FIFOs are modeled as *channels*, which encapsulate low-level details of the data transmission. *Transactions* are modeled by calling interface functions of those channels. TLM focuses more on the functionality of the data transmission and less on its actual implementation. This approach makes it easier to experiment, for example, with different busses without having to rewrite communication components.

During the design process engineers usually use a number of intermediate models. Since the models can be simulated and estimated, the results of each of the design stages can be verified. The models are classified using the *system modeling graph* (see Fig. 1) [2]. X-axis represents computation (functionality), while y-axis corresponds to communication. On each axis, there are three degrees of time accuracy (abstraction levels): *untimed*, *timed* (approximate timed), and *cycle-accurate*.

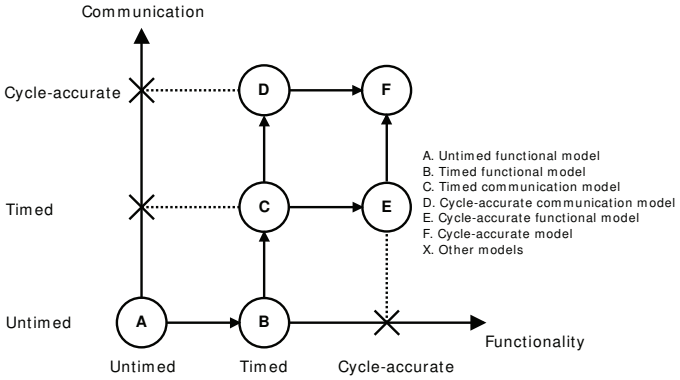


Figure 1. System modeling graph

According to [2], there are six main abstraction levels: (A) *untimed functional models*, (B) *timed functional models*, (C) *timed communication models*, (D) *cycle-accurate communication models*, (E) *cycle-accurate functional models*, and (F) *cycle-accurate models* [1]. Table 1 summarizes the characteristics of the abstraction levels mentioned.

TABLE I. CHARACTERISTICS OF THE ABSTRACTION LEVELS

Abstraction levels	Communication time	Computation time	Communication scheme
Untimed functional model (A)	No	No	Variable / Abstract channel
Timed functional model (B)	No	Approximate	Message-passing channel (FIFO)
Timed communication model (C)	Approximate	Approximate	Abstract bus channel (arbiter)

[1] The terminology we use here slightly differs from [2].

Cycle-accurate communication model (D)	Accurate	Approximate	Detailed bus channel (protocol)
Cycle-accurate functional model (E)	Approximate	Accurate	Abstract bus channel (arbiter)
Cycle-accurate model (F)	Accurate	Accurate	Wire (pin-accurate)

IV. REACTION CHECKER ARCHITECTURE

In this paper, we consider modeling and verification of single components of hardware designs. It is done deliberately to abstract away from the inner-design communication. Speaking about testbench development (more specifically, about reaction checker organization), the main interest is in the outer-design communication (interaction with the testbench environment). The generalized structure of a testbench is shown in Fig. 2.

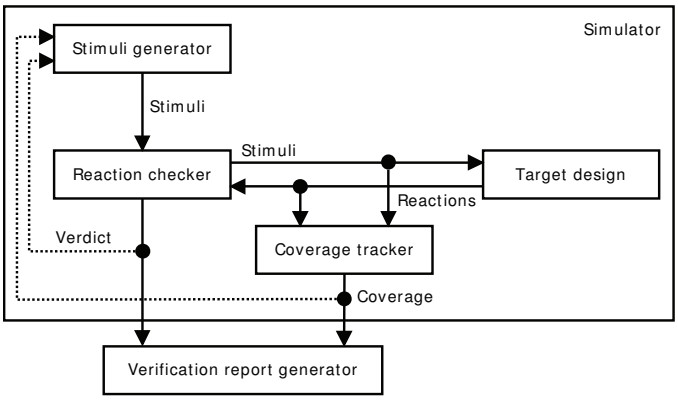


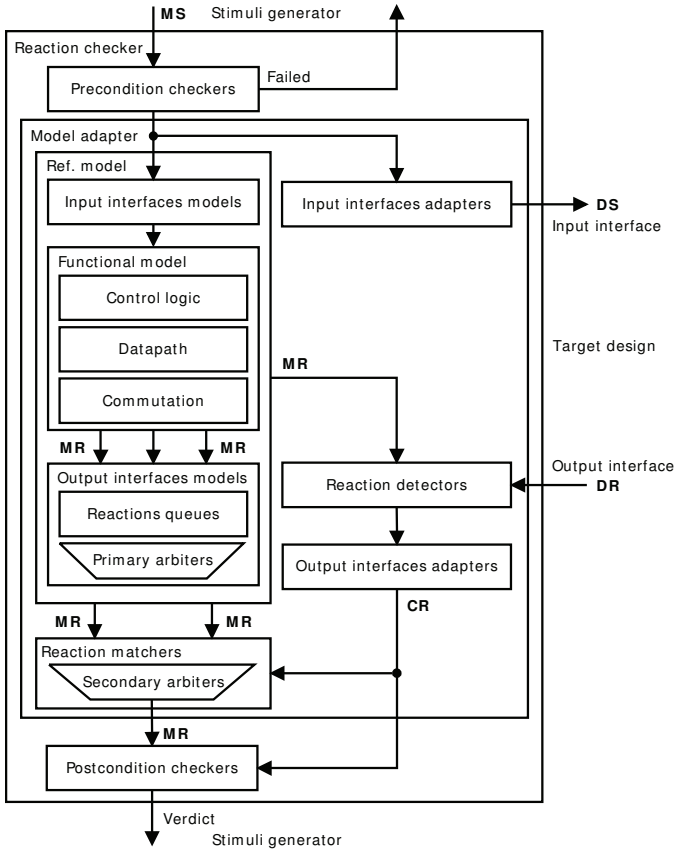
Figure 2. Generalized structure of a testbench

The stimuli generator creates a test sequence and gives the stimuli to the reaction checker which, in turn, transfers them into the design representation and applies to the target design. Reactions of the design are passed to the reaction checker, which estimates their correctness basing on assertions or a reference model. Stimuli and reactions are also given to the coverage tracker, which evaluates testing completeness using some metrics or heuristics.

This scheme clearly demonstrates the general issues of the simulation-based verification. However, the real-life is much more difficult. Design projects constantly evolve, and serious efforts are needed to make the testbench components be consistent with the design's changes. The most labor-intensive maintenance is required for reaction checkers, because they describe the reference behavior. We have formulated the following requirements for the reaction checkers constructing approaches.

- Possibility of using abstract reference models for verification of hardware components at the register transfer level (RTL).
- Easy adaption of a reaction checker to modifications of the input or output interfaces of the component being verified.
- Easy adaption of a reaction checker to changes of timing properties of the design.
- Possibility of refining reference models up to a cycle-accurate level.
- Provision of as much precise error diagnostics as it is possible for a given abstraction level.

To fulfill the requirements stipulated above, we suggest the following reaction checker's architecture (see Fig. 3). A reaction checker is decomposed into several parts, among which there are *pre-* and *postcondition checkers*, a *reference model*, and a *model adapter*. In turn, a reference model includes *input* and *output interfaces models* and a *functional model* of the target design, while a model adapter consists of *input* and *output interfaces adapters*. For each of the output interfaces there are also some components intended for reaction detection (a *reaction detector*) and for bus arbitration (*a primary arbiter*, which is a part of the reference model, and *a secondary arbiter* or *reaction matcher*, which is a part of the model adapter). The suggested architecture is TLM-compliant, because there is an explicit separation of communication (input and output interfaces models) from computation (functional model).



MS - Model stimulus (abstract message)
DS - Design stimulus (cycle- and pin-accurate serialization of MS)
MR - Model reaction (reference message or constraint)
DR - Design reaction (cycle- and pin-accurate series)
CR - Checked reaction (deserialization of DR)

Figure 3. Reaction checker's architecture

Let us consider how a reaction checker works. When a stimulus (**MS**, **Model Stimulus** in Fig. 3) is received, the corresponding precondition is checked. If there is a situation when the design's behavior is undefined (for example, if the

input protocol is violated), then the precondition fails and the stimulus is skipped; otherwise it is passed to the model adapter, which applies it both to the reference model and to the DUV. In the second case, the input interface adapter is used, which *serializes* the abstract message representing the stimulus into the concrete series of input signals (**DS**, **Design Stimulus**). The reference model emulates processing of the stimulus at some level of abstraction, calculates references reactions (in the explicit form or in the form of constraints[2]) (**MR**, **Model Reaction**), and puts them to the reactions queues of the corresponding output interface models.

When the target design's reaction is detected on an output interface (**DR**, **Design Reaction**), it is *deserialized* into the model representation (**CR**, **Checked Reaction**). Then, the reaction matcher tries to find a model reaction corresponding to that one. It requests the primary arbiter, which calculates a set of candidates. In some situations, there is exactly one reaction, but in the general case, when the reference model is rather abstract, the primary arbiter cannot choose a reaction using only information provided by the model (for example, when the reference model expects two reactions on the interface, but their order is undefined). In such cases, the secondary arbiter is used, which is given with a hint on how the model reaction should look like (it tries to match the reference reaction that is similar to the detected one).

Given a set of candidate reactions $\mathbf{C}=\{\mathbf{MR}_i\}$ (provided by the primary arbiter), a reaction **CR** to be checked, and a hint function **h**, the reaction matcher works as follows. If the set **C** is empty, then it fails. If **C** contains exactly one element, then the reaction matcher returns that element; otherwise it calculates $\mathbf{h}(\mathbf{CR})$ (it is usually a message field, checksum or message in whole) and tries to find a model reaction $\mathbf{MR}\in\mathbf{C}$ such that $\mathbf{h}(\mathbf{MR}) = \mathbf{h}(\mathbf{CR})$. If there are no such reactions, then it fails. If there is exactly one reaction, then it returns that reaction. If there are multiple reactions, then it selects randomly one of them. When some reaction is matched, it is removed from the reactions queue of the output interface.

As soon as the reaction is selected, the appropriate postcondition is checked to estimate correspondence between the design's reaction (**CR**) and the selected one (**MR**). It should be noticed that in the majority of cases equality $\mathbf{CR} = \mathbf{MR}$ is tested. If the postcondition is violated, then the reaction checker reports the failure and provides the detailed diagnostics; otherwise the process continues.

Using the described arbitration mechanism the approach makes it possible to utilize abstract models for verification of cycle- and pin-accurate RTL components. It easily copes with changes of input and output interfaces with the help of interface adapters. What is also important, it is applicable to different abstraction levels and allows refining timing properties of a reaction checker up to a cycle-accurate level (see "*Application of the approach at different abstraction levels*"). Finally, the reaction matching algorithm provides a basis for high-quality error diagnostics (see "*Classification of errors*").

[2] Constraints (particularly, undefined values) are used if the design specification admits several correct alternatives for the reaction.

V. Applications of the Approach at Different Abstraction Levels

According to the reaction checking scheme, there are two main parameters that determine abstractness of a reference model (and a reaction checker in whole): (1) *reaction detection mechanism* and (2) *reaction arbitration mechanism*. The first of them decides if the target design produces a reaction on a given output interface or not. If there is a reaction, then the second mechanism finds a model reaction for that one.

Reaction detection mechanism for a single output interface can be described as a Boolean function depending on the reference model's state ($\mathbf{S}\in\boldsymbol{MS}$) and the design's outputs ($\mathbf{O}\in\boldsymbol{DO}$)[3]:

$$\mathbf{d}: \boldsymbol{MS} \times \boldsymbol{DO} \rightarrow \{true,\ false\}.$$

Roughly speaking, **d**-functions are calculated at every simulation cycle. If one of them returns *true*, then a reaction is detected. After that, the arbitration mechanism is launched for the corresponding output interface.

Reaction arbitration mechanism (including primary and secondary arbiters) is modeled as a function that returns a model reaction or a special value *failed* depending on the model's state and a detected reaction ($\mathbf{CR}\in\boldsymbol{MR}$):

$$\mathbf{a}: \boldsymbol{MS} \times \boldsymbol{MR} \rightarrow \boldsymbol{MR} \cup \{failed\}.$$

Using **d**- and **a**-functions one can define the basic abstraction levels mentioned above.

In *cycle-accurate models*, reaction detection does not use the design's outputs. A reference model itself determines points of time when reactions should occur. In other words, all the **d**-functions of the model look as follows:

$$\mathbf{d}: \boldsymbol{MS} \rightarrow \{true,\ false\}.$$

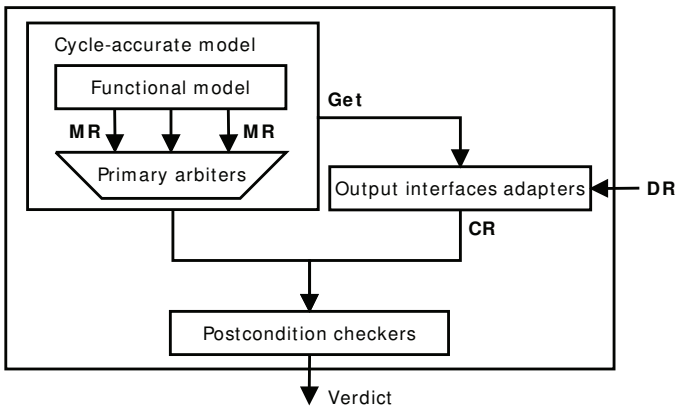


Figure 4. Reaction checker of a cycle-accurate model

Another distinctive feature of cycle-accurate models is that such models do not use secondary arbiters. Primary arbiters (which are a part of a model) are able to choose a single reaction to be compared with the detected one. This property means that all the **a**-functions do not depend on a detected reaction and do not fail:

[3] Since we usually use the black-box approach, the inner-design state is not considered here.

$$\mathbf{a}: \boldsymbol{MS} \rightarrow \boldsymbol{MR}.$$

Fig. 4 shows a response checker's fragment for a cycle-accurate model. You can see that reaction detectors and secondary arbiters are absent.

In contrast to cycle-accurate models, *untimed functional models* do not specify any timing properties of hardware components. Particularly, it implies that they do not have primary arbiters and thereby reaction matching is done with the help of secondary arbiters only. To determine points of time when reactions occur, untimed models (as all kinds of approximate timed models) use reaction detectors. **d**-functions of such models usually depend only on the design's outputs:

$$\mathbf{d}: \boldsymbol{DO} \rightarrow \{true,\ false\}.$$

A response checker for an untimed model is illustrated in Fig. 5. You can see that primary arbiters are removed and all model reactions are passed to secondary arbiters directly.

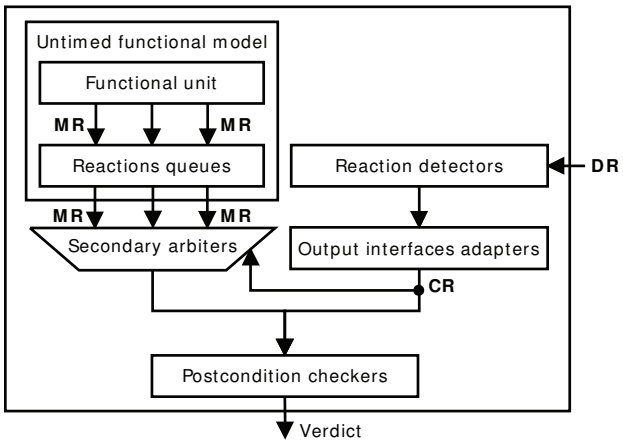


Figure 5. Reaction checker for an untimed model

Reference models of the other abstraction levels (*timed functional models*, *timed communication models*, etc.) are situated between cycle-accurate and untimed models. Since they are not very accurate, they use reaction detectors. However, as opposed to pure untimed models, timed models employ primary arbiters. The distinction between intermediate models consists mostly in different concretization of the arbiters. There are two basic classes of arbiters: *deterministic* (*order-accurate*), which choose no more than one reaction from a given set, and *nondeterministic* (*order-inaccurate*), which might select several reactions failing to determine the exact order. If nondeterminism is bounded (in a sense) we can identify *an order-approximate* class of arbiters.

Let us say few words about transitions between different abstract levels. At the beginning of the design process, when the requirements are sufficiently incomplete, only untimed functional models (**A** in Fig. 1) are applicable. As soon as the requirements are clarified, the models are concretized as well. The leading role in reusing a testbench is assigned to the communication model (to the primary arbiters in particular). In fact, communication aspects are usually more changeable.

In most of the cases approximate-timed models (**B** and **C**) are enough to achieve high-quality verification. However, in some cases (for critical components with complex control logic) cycle-accurate models (**D**, **E**, and **F**) are required [7]. It should be emphasized that changing level of a model from approximate-timed to cycle-accurate is difficult because it involves rather sophisticated computation and communication issues. The thing is that it is required only for a little number of components and only at the final design stages (i.e. not too often). In other cases, timing refinement is generally connected with the tuning of the primary arbiters.

Let us see a trivial example in C++ containing a part of a reference model that describes some operation of some DUV:

```
void DUV::Operation(Interface &input, Message &stimulus) {
  // Apply the stimulus via the input interface adapter
  RECV(input, stimulus);
  // Compute the reactions at some abstraction level
  ...
  // Inform the testbench on the expected reaction
  SEND(output, reaction);
  ...
}
```

The operation is modeled as a method with two parameters: (1) an *input interface* and (2) a *stimulus message*. The first thing the method does is call `RECV` that applies the stimulus to the inputs. After that, the reference reactions are computed and stored by calling `SEND`. In cycle-accurate models, one can use a construct `CYCLE` to emulate the DUV's cycle. Calling `SEND` in such models results in immediate reading of the design reaction from the outputs and then comparing it with the reference value. For untimed and approximate-timed models a separate process is created that, first, detects a design reaction, then, reads it, and, finally, matches it with one of the model reactions expected on the corresponding interface.

VI. CLASSIFICATION OF ERRORS

The suggested scheme for the reaction checking provides a good foundation for the error classification and diagnostics. To classify different types of bugs in hardware components, we introduce a *reaction checking graph* illustrating possible alternatives in the checking process (see Fig. 6).

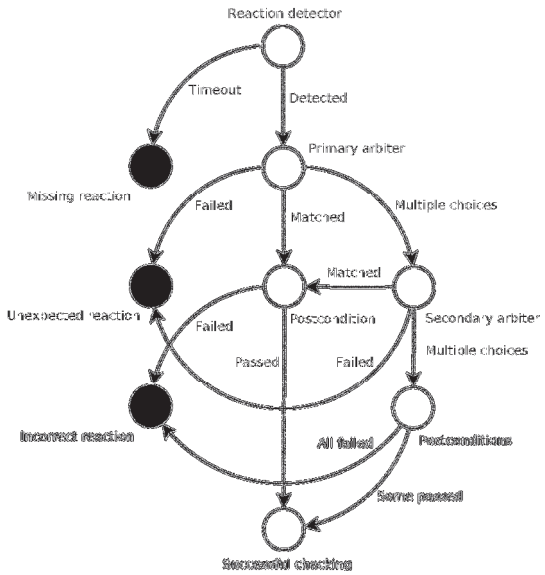


Figure 6. Reaction checking graph

The graph shown in Fig. 6 is a bit simplified, but it clearly demonstrates the main classes of errors (they are shown as black circles). If a testbench expects some reaction, but it is not detected within a reasonable amount of time, then a *missing reaction* error is reported and provided with detailed information on the expected reaction (message, output interface, and estimated time). Every time when a reaction is detected, the primary and the secondary arbiters are requested. If they both fail, then the testbench reports an *unexpected reaction* error informing about the interface and time. If the arbiters provide one-to-one matching, then the selected model reaction is compared with the detected one; otherwise the testbench warns about *nondeterministic behavior* and compares all the reactions chosen with the detected one. If there is at least one coincidence, then the reaction checking is supposed to be successful; otherwise the testbench report an *incorrect reaction* error and displays the most probable candidates for the detected reaction.

VII. EXPERIENCE

The suggested approach for developing testbenches has been used in a number of industrial projects on hardware verification (mostly for individual units of microprocessors [8]). It has demonstrated the ability to detect high-quality errors in hardware designs (including hardly detectable bugs in control logic), acceptable man-hours, and reusability of testbench components. Our experience is summarized in Table 2.

TABLE II. APPLICATIONS OF THE SUGGESTED APPROACH

Design under verification	Maximal abstractness	Minimal abstractness	Design stages
Translation lookaside buffer (TLB)	Detailed-timed model	Cycle-accurate model	Late/final
Floating point unit (FPU)	Untimed model	—	Late/final
Non-blocking L2 cache	Approximate-timed model	Detailed-timed model	Middle/late
Northbridge data switch	—	Cycle-accurate model	Final
Memory access unit (MAU)	Untimed model	Cycle-accurate model	Early/middle
System interrupt controller	Untimed model	Approximate-timed model	Early/middle

Fig. 7 demonstrates averaged labor-costs for developing reference models of hardware components at different abstraction levels. Of course, it is rather crude approximation, but it works well in real practice. For example, it has taken us about 100% of the initial efforts to add approximate timed properties to the untimed model of the system interrupt controller (1 man-month). The properties we have added describe the order of reactions on two output interfaces. It has allowed us to find an additional bug in the design (1 of 8). Another example is TLB. We have spent 1 man-week of 3 to make the reference model to be cycle-accurate and have detected 3 additional bugs of 10.

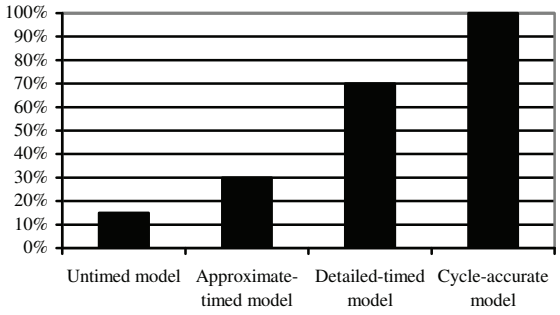


Figure 7. Relative labor-costs for developing reference models at different abstraction levels

VIII. CONCLUSIONS

It is obvious that usage of more concrete models allows detecting more complex bugs. It is also obvious that development of more concrete models requires more resources. Choosing right abstraction level for verification is a problem which is really hard to formalize. Decision making should take into account lots of factors, like application domain of the hardware being designed, available resources and many others. Regardless of the decision, it would be rather useful to have an approach making it possible to reuse the available models – refine them without changing it considerably.

The article considers the problems of developing reusable testbenches and suggests the approach that is based on the TLM concept. It allows using abstract reference models for the RTL verification and adapting reaction checkers to changes of interfaces and timing properties of the designs. In the future we are planning to improve the error diagnostics to detect such errors as reaction missequencing and demultiplexing failures.

REFERENCES

[1] J. Bergeron. “Writing testbenches: functional verification of HDL models”. Kluwer Academic Publishers, 2000.

[2] L. Cai, D. Gajski. “Transaction level modeling: an overview”. In Proc. The International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS), 2003, pp. 19-24.

[3] W. Lam. “Hardware design verification: simulation and formal method-based approaches”. Prentice Hall, 2005.

[4] C.-M.R. Ho. “Validation tools for complex digital designs”. PhD thesis, Stanford University, 1996.

[5] H.D. Foster, A.C. Krolnik, D.J. Lacey. “Assertion-based design”. Kluwer Academic Publishers, 2004.

[6] OVM User Guide – http://www.ovmworld.org.

[7] M. Chupilko, A. Kamkin. “Developing cycle-accurate contract specifications for synchronous parallel-pipeline hardware: application to verification”. In Proc. The Baltic Electronic Conference (BEC), 2010, pp. 185-188.

[8] M. Chupilko, A. Kamkin, D. Vorobyev. “Methodology and experience of simulation-based verification of microprocessor units based on cycle-accurate contract specifications”. In Proc. The Spring Young Researchers’ Colloquium on Software Engineering (SYRCoSE), 2008, vol. 2, pp. 25-31.