# Runtime Verification Based on Executable Models:
# On-the-Fly Matching of Timed Traces

Mikhail Chupilko        Alexander Kamkin

Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
109004, Russia, Moscow, Alexander Solzhenitsyn st., 25.

{chupilko,kamkin}@ispras.ru

Runtime verification is checking whether a system execution satisfies or violates a given correctness property. A procedure that automatically, and typically on the fly, verifies conformance of the system's behavior to the specified property is called a monitor. Nowadays, a variety of formalisms are used to express properties on observed behavior of computer systems, and a lot of methods have been proposed to construct monitors. However, it is a frequent situation when advanced formalisms and methods are not needed, because an executable model of the system is available. The original purpose and structure of the model are out of importance; rather what is required is that the system and its model have similar sets of interfaces. In this case, monitoring is carried out as follows. Two "black boxes", the system and its reference model, are executed in parallel and stimulated with the same input sequences; the monitor dynamically captures their output traces and tries to match them. The main problem is that a model is usually more abstract than the real system, both in terms of functionality and timing. Therefore, trace-to-trace matching is not straightforward and allows the system to produce events in different order or even miss some of them. The paper studies on-the-fly conformance relations for timed systems (i.e., systems whose inputs and outputs are distributed along the time axis). It also suggests a practice-oriented methodology for creating and configuring monitors for timed systems based on executable models. The methodology has been successfully applied to a number of industrial projects of simulation-based hardware verification.

## 1    Introduction

*Verification* has long been recognized as one of the integral parts of software and hardware design processes [15, 22]. Generally, it is an activity intended to check whether a system or its part meets a *specification* (set of functional and timing requirements). Verification techniques can be divided into two main groups, namely *formal verification* and *testing* (also known as *simulation-based verification* in the hardware engineering domain) [14]. Formal methods are aimed at rigorous proving or disproving the correctness of a formal model of a system with respect to a formal specification. Such approaches exhaustively examine all possible executions of a given system – either explicitly (by enumerating all reachable states) or implicitly (by using symbolic techniques). In contrast, testing deals with a finite number of executions and estimates the system's behavior in a finite number of situations (so-called *test situations*). *Runtime verification* is a common point of both. Like testing, it works with concrete executions of a system, but does it in a formal way.

In runtime verification, a correctness property is typically expressed in a formal language, which makes it possible to automatically translate the property into a *monitor*. Such a monitor is then used to check a system execution with respect to that property [5]. The idea is similar to *specification-based testing*, where a formal specification serves as a basis for generating a *test oracle*, which, like monitor, determines whether an observed behavior is consistent with the specification [11, 12]. But, as opposed to testing, it is not a scope of runtime verification to construct test sequences and apply them to the

system under test. The task is to passively observe inputs and outputs of the system and to check their conformance – that is why it is also called *passive testing* [3]. Formally, when $\mathcal{L}(\varphi)$ denotes the set of valid system executions given by property $\varphi$, runtime verification is aimed at checking whether a concrete execution $w$ is an element of $\mathcal{L}(\varphi)$. In this sense, runtime verification deals with the *word problem*, i.e., identifying whether a given word is included in some language [5].

Correctness properties in runtime verification may be expressed using a variety of different formalisms, including *extended regular expressions* [21], *contract specifications* [12] and *rule-based approaches* [4]. *Temporal logic*, which is well-known from *model checking* [10], is also very popular in runtime verification, especially variants of *linear temporal logic*, such as LTL and TLTL (a natural counterpart of LTL in the timed setting) [5]. There are also a lot of methods for generating effective monitors (or test oracles) from formal specifications. However, sophisticated formalisms and methods are not always suitable for industrial practice. For example, many hardware design companies use *executable software models* for design space exploration and architecture modeling; it is quite natural to *reuse* those models for verification and monitoring. High reusability within a project is important to complete verification within the timeline [19]. Moreover, reusable models ensure conceptual integrity of the project and accelerate the knowledge interchange.

Runtime verification based on executable models is carried out in the following way. A reference model is *co-executed* with the target system and applied with the same inputs as the system under verification. The outputs of two "black boxes" are given to the monitor that matches them and decides whether they are consistent. Aside from minor technical difficulties on organizing co-execution and transforming interfaces, there is a conceptual problem relating to *model abstractness*. As a rule, a model (tending to be as simple as possible) does not specify the system's behavior accurately, which makes the output matching awkward. If the model produces some outputs in some order, it does not necessarily mean that the system should do it in the same manner – the order may differ and some of the ouputs may be omitted. Before using a model for monitoring one has to specify a priori information on its abstractness and give it to the monitor. One of the contributions of this paper is an approach that allows easy adaptation of monitors for models represented in different abstraction levels.

We consider *timed systems*, which react on inputs distributed in time and emit outputs at dedicated time points. Formally, it means that each event is paired together with a time stamp, identifying when exactly the event happened. For the *discrete-time model*, timed sequences of events can be easily transformed into ordinary ones by removing time stamps and inserting a special *tick* event in proper positions of the original sequence (as many times as necessary) [2]. Nevertheless, even in that case, it is convenient to suppose that each event is tagged with a time stamp. Executions of a system and its model are described by *timed sequences* over the same alphabet. Assumptions on the model abstractness allow dynamical generalization of linear sequences into the *partially ordered multisets* consisting of events and *time intervals* associated with them. In general terms, the monitor checks on the fly that an implementation trace is a linearization of the generalized specification trace (subset of the trace) and all implementation events satisfy the corresponding time interval constraints.

The rest of the paper is organized as follows. Section 2 introduces the basic mathematical notions used in the work such as a *timed word*, *trace* and *pomset*. Section 3 is the main part of the paper, in which the suggested method for timed trace matching is described. The section formalizes implementation and specification behavior and defines a conformance relation between implementations and specifications. It also describes a monitoring approach in detail and states its correctness. Section 4 outlines our experience in using the proposed approach for simulation-based verification of industrial hardware designs. Section 5 is a brief survey of the related work. Section 6 concludes the paper and discusses some of our future research directions.

## 2 Preliminaries

For the rest of the paper, $\Sigma$ denotes a finite alphabet of *events*, while $\mathbb{T}$ denotes a *time domain*. An event might be considered as a set of propositions that identify a situation when the event happens. A time domain is a totally ordered set with no upper bound, typically, $\mathbb{N}$ (discrete-time model) or $\mathbb{R}^{\geq 0}$ (continuous-time model). Sequences of events are called *words* (the *empty word* is denoted by $\varepsilon$). Symbols $\Sigma^*$ and $\Sigma^{\omega}$ stand for the sets of *finite* and *infinite* words over $\Sigma$, respectively. The length of a word $w$ is denoted by $|w|$. If $u$ and $v$ are two words over the same alphabet and $u$ is finite, then $uv$ denotes their *concatenation*. For $w = uv$ we say that $w$ is a *continuation* of $u$ with $v$.

Sometimes, it is useful to structurize events by dividing them into *inputs* and *outputs* ($\Sigma = I \cup O$) and by introducing a notion of *port* [17]. Let $P = \{1, 2, ..., k\}$ and $\text{port} : \Sigma \to P$. Then, the tuple $\langle \Sigma_1, ..., \Sigma_k \rangle$, where $\Sigma_p = \text{port}^{-1}(p)$, is called a *distributed alphabet*.

**Definition 1 (Timed word – Alur and Dill [2])** *A timed word $w$ over the alphabet $\Sigma$ and the time domain $\mathbb{T}$ is a sequence $(a_0, t_0)(a_1, t_1)...$ of timed events $(a_i, t_i) \in \Sigma \times \mathbb{T}$, satisfying the following constraints:*

   1. *for each $i \geq 0$, $t_i < t_{i+1}$ holds (monotonicity);*

   2. *if the sequence is infinite, for every $t \in \mathbb{T}$ there is some $i \geq 0$, such that $t_i > t$ (progress).* □

Strict monotonicity in the definition above can be weaken to monotonicity (i.e., it can be required that $t_i \leq t_{i+1}$ for all $i \geq 0$) [2]. $(\Sigma \times \mathbb{T})^*$ and $(\Sigma \times \mathbb{T})^{\omega}$ denote the sets of finite and infinite timed words, respectively. Note that port partitioning implies an additional constraint on a timed word:

   3. *for all $i, j \geq 0$, such that $i \neq j$ and $t_i = t_j$, $\text{port}(e_i) \neq \text{port}(e_j)$ (sequentiality).*

In concurrent systems, the concept of *independence* is often in use. Two events are considered as *independent* if they cannot be causally related (i.e., they may happen concurrently). Events on different ports are usually independent, while those on the same port are dependent. Concurrent execution can be modeled by partially ordered traces of events, where incomparable events are supposed to occur in indeterminate order or in parallel [6]. This intuition underlies two formal models of non-interleaving concurrency: (1) *Mazurkiewicz's trace model* [18] and (2) *Pratt's pomset model* [20]. The definitions and their extensions for the timed case are given below.

**Definition 2 (Trace – Mazurkiewicz [18])** *An independence relation over the alphabet $\Sigma$ is a symmetric and irreflexive relation $\mathbb{J} \subset \Sigma \times \Sigma$. Given an independence relation $\mathbb{J}$, a pair $\langle \Sigma, \mathbb{J} \rangle$ is called a concurrent alphabet. Two words $u$ and $v$ are called Mazurkiewicz equivalent ($u \equiv_{\mathbb{J}} v$) iff $u$ can be transformed to $v$ by a finite number of exchanges of adjacent, independent events. A Mazurkiewicz trace (or, simply, a trace) is an equivalence class of words by the Mazurkiewicz equivalence relation.* □

The set of traces over the concurrent alphabet $\langle \Sigma, \mathbb{J} \rangle$ is denoted as $\mathbb{M}(\Sigma, \mathbb{J})$. Given an independence relation $\mathbb{J}$, the relation $\mathcal{D} = (\Sigma \times \Sigma) \setminus \mathbb{J}$ is called the *dependence relation*. The *length* of a trace $\tau$ (denoted by $|\tau|$) is the length of any of its representatives. If $w$ is a word, $[w]_{\mathbb{J}}$ is the trace that includes $w$ as a representative. A *concatenation* of traces over the same concurrent alphabet $\langle \Sigma, \mathbb{J} \rangle$ is defined by the equality $[u]_{\mathbb{J}}[v]_{\mathbb{J}} = [uv]_{\mathbb{J}}$. A trace $\sigma$ is called a *prefix* of $\tau$ ($\sigma \sqsubseteq \tau$) iff there exists $\gamma$, such that $\sigma\gamma = \tau$.

**Example 1 (Traces)** *Let $\Sigma = \{a, b, c, d\}$ and $\mathbb{J} = \{(a, b), (b, a), (c, d), (d, c)\}$. Then, some traces are as follows:*

$$
\begin{aligned}
[\varepsilon]_{\mathbb{J}} &= \{\varepsilon\} \\
[ad]_{\mathbb{J}} &= \{ad\} \\
[ab]_{\mathbb{J}} &= \{ab, ba\} \\
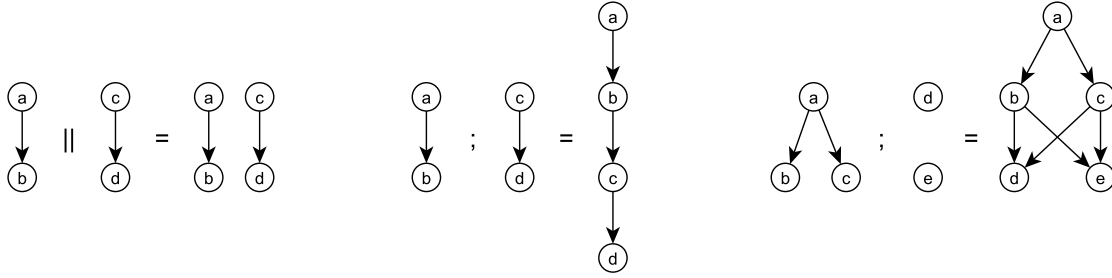[abcd]_{\mathbb{J}} &= \{abcd, bacd, abdc, badc\}
\end{aligned}
$$

Figure 1: Sequential and parallel composition of simple pomsets

**Definition 3 (Pomset (partially ordered multiset) – Pratt [20])** *A $\Sigma$-labeled partial order is a tuple $\langle V, \preceq, \lambda \rangle$, where $V$ is a finite set of vertices and $\lambda : V \to \Sigma$ is the labeling function. Two $\Sigma$-labeled partial orders are called equivalent iff they are order- and label-isomorphic (i.e., they are either equal or differ only in the names of vertices). A pomset over the alphabet $\Sigma$ is an isomorphism class of $\Sigma$-labeled partial orders.* □

Note that words are equivalent to pomsets with the total order, while multisets are equivalent to pomsets whose partial order is the equality. For convenience, we will use a concrete representative (a labeled partial order) to denote the pomset. There is a number of operations on pomsets, including *parallel* and *sequential composition*. Let $\sigma = \langle V, \preceq, \lambda \rangle$ and $\gamma = \langle V', \preceq', \lambda' \rangle$ are pomsets over the same alphabet, such that $V \cap V' = \varnothing$. Define the pomsets $(\sigma \parallel \gamma)$ and $(\sigma \,;\, \gamma)$ as follows:

$$
\begin{aligned}
(\sigma \parallel \gamma) &= \langle V \cup V', \preceq \cup \preceq', \lambda \cup \lambda' \rangle \\
(\sigma \,;\, \gamma) &= \langle V \cup V', \preceq \cup \preceq' \cup (V \times V'), \lambda \cup \lambda' \rangle
\end{aligned}
$$

**Example 2 (Pomsets)** *Examples of pomsets in the form of Hasse diagrams (i.e., drawings of the partial order transitive reduction), may be found in Figure 1.*

A *linearization* of a pomset $\langle V, \preceq, \lambda \rangle$ is a total labelled order $\langle V, \leq, \lambda \rangle$, where $\preceq \subseteq \leq$. The set of linearizations of a pomset $\sigma$ is denoted by $\mathsf{lin}(\sigma)$. A designation $x \perp y$ means that neither $x \preceq y$ nor $y \preceq x$. We say that $x \in V$ *immediately precedes* $y \in V$ and write $x \mathbin{\dot\prec} y$ iff $x \prec y$ and there is no such $z \in V$ that $x \prec z \prec y$. A *history* of $x \in V$ is the set $\downarrow x = \{ y \in V \mid y \preceq x \}$ (for $X \subseteq V$, $\downarrow X = \bigcup_{x \in X} \downarrow x$).

It can be shown that each trace can be represented as a pomset. The opposite is true only for a restricted class of pomsets [6]. Let $\langle \Sigma, \mathfrak{I} \rangle$ be a concurrent alphabet and $\sigma = \langle V, \prec, \lambda \rangle$ be a pomset, such that

- for each $x \in V$, $\downarrow x$ is a finite set;

- for all $x, y \in V$, if $x \perp y$, then $\big( \lambda(x), \lambda(y) \big) \in \mathfrak{I}$;

- for all $x, y \in V$, if $x \mathbin{\dot\prec} y$, then $\big( \lambda(x), \lambda(y) \big) \in \mathfrak{D}$.

Then, $\mathsf{lin}(\sigma)$ is a trace over $\langle \Sigma, \mathfrak{I} \rangle$ and $\sigma = \mathsf{pom}(\mathsf{lin}(\sigma))$ [6]. Further, we will represent traces as pomsets satisfying the conditions above. The same consideration is done in [16, 8].

**Definition 4 (Timed trace – Chieu and Hung [8])** *A timed trace over the concurrent alphabet $\langle \Sigma, \mathfrak{I} \rangle$ and the time doman $\mathbb{T}$ is a quadruple $\langle V, \preceq, \lambda, \theta \rangle$, where $\langle V, \preceq, \lambda \rangle$ is a trace over $\langle \Sigma, \mathfrak{I} \rangle$ and $\theta : V \to \mathbb{T}$ is a time function satisfying the following conditions:*
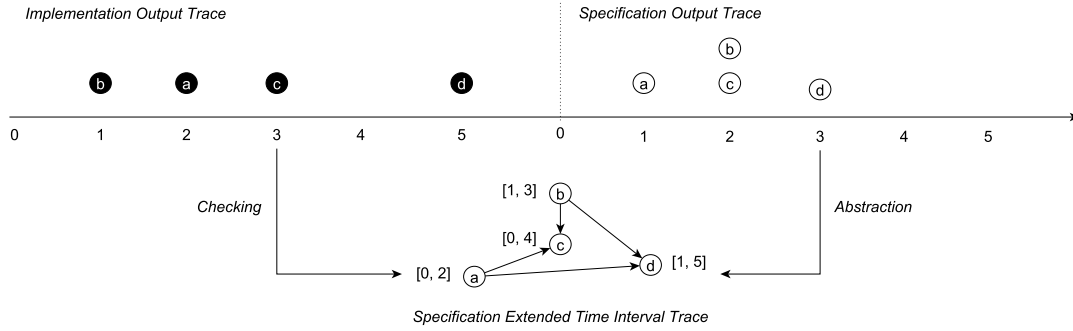
Figure 2: Scheme for checking conformance between implementation and specification

1. *for all $x, y \in V$, if $x \prec y$, then $\theta(x) < \theta(y)$ (causality);*

2. *if the trace is infinite, then for every $t \in \mathbb{T}$ there is a cut $C \subseteq V$, such that $\min_{x \in C}\{\theta(x)\} \geq t$ (progress).* □

The set of timed traces over the concurrent alphabet $\langle \Sigma, \mathfrak{I} \rangle$ and the time domain $\mathbb{T}$ is denoted as $\mathbb{M}_\theta(\Sigma, \mathfrak{I}, \mathbb{T})$. Note that timed words are a particular case of timed traces. Given a non-empty timed trace $\sigma = \langle V, \preceq, \lambda, \theta \rangle$, $\mathsf{begin}(\sigma) = \min_{x \in V}\{\theta(x)\}$ and $\mathsf{end}(\sigma) = \max_{x \in V}\{\theta(x)\}$ (if $\sigma$ is infinite, $\mathsf{end}(\sigma) = \infty$); $\sigma_{[t, t+\Delta t]}$ is a sub-trace of $\sigma$ consisting of $x \in V$, such that $\theta(x) \in [t, t+\Delta t]$. Let $\mathfrak{TI}(\mathbb{T})$ be the set of time intervals over the time domain $\mathbb{T}$ (i.e., $\mathfrak{TI}(\mathbb{T}) = \{[t, t+\Delta t] \mid t, t+\Delta t \in \mathbb{T}\}$).

**Definition 5 (Time interval trace)** *A time interval trace over the concurrent alphabet $\langle \Sigma, \mathfrak{I} \rangle$ and the time doman $\mathbb{T}$ is a quadruple $\sigma = \langle V, \preceq, \lambda, \delta \rangle$, where $\langle V, \preceq, \lambda \rangle$ is a trace over $\langle \Sigma, \mathfrak{I} \rangle$ and $\delta : V \to \mathfrak{TI}(\mathbb{T})$ is a function that associates a time interval to a vertex. The language of the time interval trace $\sigma$ is the set $\mathcal{L}(\sigma) = \{\langle V, \preceq, \lambda, \theta \rangle \in \mathbb{M}_\theta(\Sigma, \mathfrak{I}, \mathbb{T}) \mid \forall x \in V . \theta(x) \in \delta(x)\}$.* □

The set of time interval traces over the concurrent alphabet $\langle \Sigma, \mathfrak{I} \rangle$ and the time domain $\mathbb{T}$ is denoted as $\mathbb{M}_\delta(\Sigma, \mathfrak{I}, \mathbb{T})$. Futher we will deal with pairs consisting of a timed trace $\sigma$ and a time interval trace $\sigma_\delta$, such that $\sigma \in \mathcal{L}(\sigma_\delta)$. Such a pair can be expressed as a quintuple $\langle V, \preceq, \lambda, \theta, \delta \rangle$ and is referred to as an *extended time interval trace*. The set of such traces is denoted as $\mathbb{M}_{\theta\delta}(\Sigma, \mathfrak{I}, \mathbb{T})$.

## 3 Runtime Verification with Executable Models

A timed word (more precisely, a timed trace with an empty partial order) describes a concrete execution of the *implementation* under verification, while an extended time interval trace being more general can be considered as a *specification* behavior. Our goal is to check whether an implementation timed word $w_I \in (\Sigma \times \mathbb{T})^{*(\omega)}$ is conforming to a specification trace $\sigma_S \in \mathbb{M}_{\theta\delta}(\Sigma, \mathfrak{I}, \mathbb{T})$. Note that we are interested in *on-the-fly* checking, which means that a monitor "lives" in time and matches two traces in an *event-driven* fashion. *Trace acceptance* (*verdict*) at a given time point has a three-valued semantics [5]: (1) *false* (an inconsistency has been detected), (2) *true* (the implementation execution has been completed and its trace is conforming to the specification trace) and (3) *inconclusive* (the monitoring is in progress and no inconsistency has been found).

To make it clear where a specification trace comes from, an additional explanation should be provided. As it was said in the introduction, a system specification is represented in the *executable* form.

Hence, it can be executed and its executions (as ones of the implementation) are represented as timed words. The straightforward testing of the equality of two timed words is often inadequate and makes sense only for *time-accurate* specifications. Specifications are usually more abstract than implementations, especially in terms of *event ordering* and *timing*. Assumptions on the specification abstractness generalize a concrete timed word to the extended time interval trace softening the conformance checking. Formally, *abstraction* is a map $\mathcal{A} : (\Sigma \times \mathbb{T})^{*(\omega)} \to \mathbb{M}_{\theta\delta}(\Sigma, \mathcal{I}, \mathbb{T})$, such that $w$ is conforming to $\mathcal{A}(w)$ for every $w \in (\Sigma \times \mathbb{T})^{*(\omega)}$. A specification timed word $w_S$ is mapped into the extended time interval trace $\mathcal{A}(w_S) = \sigma_S$. Then, it is checked whether an implementation word $w_I$ is conforming to the constructed specification trace $\sigma_S$. This scheme is illustrated in Figure 2. Technical details can be found in Section 4.

## 3.1    Conformance Relation

The next definition formalizes system executions in terms of timed traces. It also singles out *input* and *output sequences* as particular cases of traces corresponding to stimuli to a system and its reactions, respectively. System behavior is then abstractly defined as a map of inputs to outputs.

**Definition 6 (Execution trace)** *An execution trace over the concurrent alphabet $\langle \Sigma, \mathcal{I} \rangle$ and the time domain $\mathbb{T}$ is a timed trace with the empty partial order (i.e., a trace of the kind $\langle V, \varnothing, \lambda, \theta \rangle$). If $\Sigma = I \cup O$, then execution traces over the alphabet $I$ are called input sequences, while execution traces over the alphabet $O$ are referred to as output sequences.* □

Note that the empty partial order in execution traces reflects a fact that an implementation is a "black box", and, therefore, the cause-effect relation between its events is unknown. The sets of input and output sequences are designated by $\mathbb{I}_{\theta}(\Sigma, \mathbb{T})$ and $\mathbb{O}_{\theta}(\Sigma, \mathbb{T})$, respectively. Hereinafter, we will use the shortened notations: $\mathbb{I} = \mathbb{I}_{\theta}(\Sigma, \mathbb{T})$ and $\mathbb{O} = \mathbb{O}_{\theta}(\Sigma, \mathbb{T})$.

**Definition 7 (Behavior)** *Deterministic timed behavior (or, simply, behavior) over the alphabet $\Sigma$ and the time domain $\mathbb{T}$ is a (partial) map $\mathcal{B} : \mathbb{I} \times \mathbb{T} \to \mathbb{O}$ satisfying the following constraints:*

- *for every $w \in \mathbb{I}$ and $t \in \mathbb{T}$, $\mathsf{end}\big(\mathcal{B}(w,t)\big) \leq t$ holds (future uncertainty);*
- *for every $w \in \mathbb{I}$ and $t \in \mathbb{T}$, $\mathcal{B}(w,t) = \mathcal{B}(w_{[0,t]}, t)$ holds (time directivity);*
- *for every $w \in \mathbb{I}$ and every $t \in \mathbb{T}$, there exists $wv \in \mathbb{I}$, a continuation of $w$, and $\Delta t \geq 0$, such that $\mathsf{end}\big(\mathcal{B}(wv, t + \Delta t)\big) \geq t$ (liveness).* □

The idea behind the concept is clear. Behavior describes how an input sequence is transformed into the output sequence taking into account an observation time point. Usually, when an input sequence is applied, then after a finite number of time units (counting from the last input time) the output sequence is fully observed and is ready to be checked. Such post-mortem analysis is not however what we are interested in. There are two reasons for that: (1) to ease the analysis, an execution should be terminated as soon as a failure is detected; (2) storing long sequences in memory is costly. Providing that a reference model is available, consider how it can be used for checking implementation behavior in runtime. Let us extend the definition above by allowing a specification to return extended time interval traces over the outputs (not concrete sequences as it is required). Denote the set of such traces as $\mathbb{O}_{\theta\delta}$.

Given an output trace $\langle V, \preceq, \lambda, \theta, \delta \rangle \in \mathbb{O}_{\theta\delta}$, define two functions, $\Delta t^{\pm}$, such that for every $x \in V$, $\delta(x) = [\theta(x) - \Delta t^-(x), \theta(x) + \Delta t^+(x)]$. Assume that functions $\Delta t^{\pm}$ are bounded (i.e., there exist constants $\Delta T^{\pm} > 0$, such that $|\Delta t^{\pm}(x)| \leq \Delta T^{\pm}$ for all $x \in V$). Assume also that values $\Delta t^{\pm}(x)$ depend only on the event not on the vertex itself (i.e., $\Delta t^{\pm}(x) = \Delta t^{\pm}(\lambda(x))$). Let $\mathcal{I}$ and $\mathcal{S}$ be an *implementation* and *specification behavior*, respectively. Given an input sequence $w \in \mathbb{I}$, a time point $t \in \mathbb{T}$, let us consider
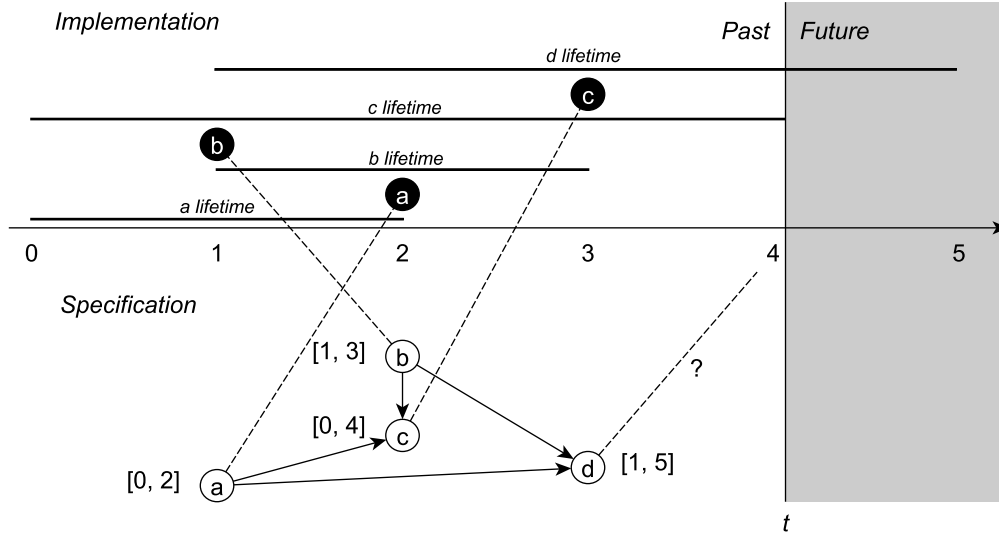
Figure 3: Conformance between implementation and specification

implementation and specification outputs: $\mathcal{I}(w,t) = \langle V_{\mathcal{I}}, \varnothing, \lambda_{\mathcal{I}}, \theta_{\mathcal{I}} \rangle$ and $\mathcal{S}(w,t) = \langle V_{\mathcal{S}}, \preceq_{\mathcal{S}}, \lambda_{\mathcal{S}}, \theta_{\mathcal{S}}, \delta_{\mathcal{S}} \rangle$. Let us introduce the following notations:

$$
\begin{aligned}
\mathsf{past}_{\mathcal{I}}^{\Delta t}(w,t) &= \{ y \in \mathcal{I}(w,t) \mid \theta_{\mathcal{I}}(y) \le (t - \Delta t^-(y)) \}; \\
\mathsf{past}_{\mathcal{I}}(w,t) &= \{ y \in \mathcal{I}(w,t) \mid \theta_{\mathcal{I}}(y) \le t \}; \\
\mathsf{past}_{\mathcal{S}}^{\Delta t}(w,t) &= \{ x \in \mathcal{S}(w,t) \mid \theta_{\mathcal{S}}(x) \le (t - \Delta t^+(x)) \}; \\
\mathsf{past}_{\mathcal{S}}(w,t) &= \{ x \in \mathcal{S}(w,t) \mid \theta_{\mathcal{S}}(x) \le t \}; \\
\mathsf{match}(x,y) &= (\lambda_{\mathcal{I}}(y) = \lambda_{\mathcal{S}}(x)) \wedge (\theta_{\mathcal{I}}(y) \in \delta_{\mathcal{S}}(x)).
\end{aligned}
$$

**Definition 8 (Conformance relation)** *The implementation behavior $\mathcal{I}$ is said to be conforming to the specification behavior $\mathcal{S}$ iff* $\mathsf{dom}\,\mathcal{I} = \mathsf{dom}\,\mathcal{S}$ *and for all $w \in \mathsf{dom}\,\mathcal{S}$ and $t \in \mathbb{T}$, there is a relation $\mathcal{M}(w,t) \subseteq \{(x,y) \in \mathsf{past}_{\mathcal{S}}(w,t) \times \mathsf{past}_{\mathcal{I}}(w,t) \mid \mathsf{match}(x,y)\}$ (called a matching relation), such that:*

1. *$\mathcal{M}(w,t)$ is a one-to-one relation;*

2. *for each $x \in \mathsf{past}_{\mathcal{S}}^{\Delta t}(w,t)$, there is $y \in \mathsf{past}_{\mathcal{I}}(w,t)$, such that $(x,y) \in \mathcal{M}(w,t)$;*

3. *for each $y \in \mathsf{past}_{\mathcal{I}}^{\Delta t}(w,t)$, there is $x \in \mathsf{past}_{\mathcal{S}}(w,t)$, such that $(x,y) \in \mathcal{M}(w,t)$;*

4. *for all $(x,y), (x',y') \in \mathcal{M}(w,t)$, if $x \prec x'$, then $\theta_{\mathcal{I}}(y) \le \theta_{\mathcal{I}}(y')$.*

*If for some $w \in \mathbb{I}$ and $t \in \mathbb{T}$ the abovementioned properties are violated, then $\mathcal{I}$ is said to be not conforming to $\mathcal{S}$, and $w_{[0,t]}$ is referred to as a counterexample.* $\square$

Figure 3 illustrates the conformance relation definition for a particular input sequence (being unimportant it is not shown in the picture) and observation time ($t = 4$). The upper part of the figure is a drawing of the implementation outputs (black circles with white labels: *b*, *a* and *c*). The lower part depicts the specification outputs (white circles with black labels: *a*, *b*, *c* and *d*). Let us denote the trace

vertices (i.e., circles themselves) by $y_b$, $y_a$ and $y_c$ (for the implementation) and $x_a$, $x_b$, $x_c$ and $x_d$ (for the specification). The implementation vertices are not causally related to each other, while the specification vertices are partially ordered (the precedence relation is drawn by arrows: $x_a \prec x_c$, $x_b \prec x_c$, $x_a \prec x_d$ and $x_b \prec x_d$) and are tagged with time intervals ($\delta(x_a) = [0,2]$, $\delta(x_b) = [1,3]$, $\delta(x_c) = [0,4]$ and $\delta(x_d) = [1,5]$). Matchings are depicted by intermittent lines connecting the implementation vertices with the specification ones ($(x_a, y_a)$, $(x_b, y_b)$ and $(x_c, y_c)$). It is easy to see that this relation fits the matching relation definition: (1) it is one-to-one relation; (2 & 3) it includes all events whose lifetime has been exhausted; (4) is preserves the specification ordering:

- $(x_a \prec x_c)$ and $(\theta(y_a) = 2 \leq 3 = \theta(y_c))$;

- $(x_b \prec x_c)$ and $(\theta(y_b) = 1 \leq 3 = \theta(y_c))$.

And, certainly, this relation satisfies the matching condition:

- $(\lambda(x_a) = \lambda(y_a) = a)$ and $(\theta(y_a) = 2 \in [0,2] = \delta(x_a))$;

- $(\lambda(x_b) = \lambda(y_b) = b)$ and $(\theta(y_b) = 1 \in [1,3] = \delta(x_b))$;

- $(\lambda(x_c) = \lambda(y_c) = c)$ and $(\theta(y_c) = 3 \in [0,4] = \delta(x_c))$.

The next section describes a procedure that automatically and dynamically constructs a matching relation between implementation and specification outputs. If it fails to create such a relation, it reports the reason, which can interpreted as a failure type: a *missing* or *unexpected* implementation output.

## 3.2   On-the-Fly Trace Matching

A monitor that matches implementation and specification traces and checks their conformance is co-executed with the implementation and specification and reacts on their outputs. Formally, the monitor can be expressed as a *timed automaton* [2] with two types of input ports: (1) ports for receiving *specification outputs* and (2) ports for receiving *implementation outputs*. When the automaton detects inconsistency between implementation and specification traces, it goes into a dedicated state informing that the implementation is not conforming to the specification.

A formal description of the on-the-fly trace matcher is given below. It is represented as a system of *guarded actions*. Each *action* is atomic and is executed as soon as the *guard* is *true*. The actions and their guards depend on an external variable $t$ reflecting the current *simulation time* and outputs produced by the specification and implementation in response to the same input sequence ($S$ and $I$, respectively). The value of $t$ is monotonically increasing in *real time* (simulation time may coincide with real time). The writing $y \in I[t]$ means that at time $t$ the implementation omits an output $y$. The description is based on two functions: (1) the *primary arbiter* ($arbiter_S$) and (2) the *secondary arbiter* ($arbiter_I$), which are defined as follows:

$$arbiter_S(X) = \begin{cases} \min_{\preceq}(X) & \text{if } X \neq \varnothing, \\ \phi & \text{otherwise } (\phi \notin \Sigma); \end{cases}$$

$$arbiter_I(y, X) = \begin{cases} \arg\min_{x \in X.\text{match}(x,y)} \{\theta_S(x)\} & \text{if there is } x \in X, \text{ such that } \text{match}(x, y), \\ \phi & \text{otherwise.} \end{cases}$$

---

**Action 1** $onSpecOutput[x], x \in S[t]$

**Guard:** *true*

**Input:** $x$

   $past_S \Leftarrow past_S \cup \{x\}$

   **if** $x \in arbiter_S(past_S)$ **then**

      **for all** $y \in past_I$ [in ascending of $\theta_{\mathtt{J}}(y)$] **do**

         **if** $x = arbiter_I(y, \{x\})$ **then**

            $past_S \Leftarrow past_S \setminus \{x\}$

            $past_I \Leftarrow past_I \setminus \{y\}$

            $match \Leftarrow match \cup \{\langle x, y \rangle\}$

            $trace(\langle x, y \rangle,$ "Conforming output")

            **break**

         **end if**

      **end for**

   **end if**

---

**Action 2** $onImplOutput[y], y \in I[t]$

**Guard:** *true*

**Input:** $y$

   $past_I \Leftarrow past_I \cup \{y\}$

   $x \Leftarrow arbiter_I(y, arbiter_S(past_S))$

   **if** $x \neq \phi$ **then**

      $past_S \Leftarrow past_S \setminus \{x\}$

      $past_I \Leftarrow past_I \setminus \{y\}$

      $match \Leftarrow match \cup \{\langle x, y \rangle\}$

      $trace(\langle x, y \rangle,$ "Conforming output")

   **end if**

---

**Action 3** $onSpecTimeout[x], x \in past_S$

**Guard:** $\left( \theta_S(x) + \Delta t^+(x) \right) \leq t$

**Input:** $x$

   $past_S \Leftarrow past_S \setminus \{x\}$

   $verdict \Leftarrow false`$

   $trace(\langle x, \phi \rangle,$ "Missing output")

   **terminate**

---

**Action 4** $onImplTimeout[y], y \in past_I$

**Guard:** $\left( \theta_{\mathtt{J}}(y) + \Delta t^-(y) \right) \leq t$

**Input:** $y$

   $past_I \Leftarrow past_I \setminus \{y\}$

   $verdict \Leftarrow false$

   $trace(\langle \phi, y \rangle,$ "Unexpected output")

   **terminate**

---

**Action 5** $onInitialize$

**Guard:** $t = 0$

**Input:** $\varnothing$

   $past_S \Leftarrow \varnothing$

   $past_I \Leftarrow \varnothing$

   $match \Leftarrow \varnothing$

---

**Action 6** $onFinalize$

**Guard:** $\left( \mathbf{end}(S) + \Delta T^+ \right) \leq t \wedge \left( \mathbf{end}(I) + \Delta T^- \right) \leq t$

**Input:** $\varnothing$

   $verdict \Leftarrow true$

   **terminate**

---

Given a time point, the timeout actions (*onSpecTimeout* and *onImplTimeout*), if they are activated, are called after the output reception actions (*onSpecOutput* and *onImplOutput*). Otherwise, there might be a *false negative*. E.g., when the implementation sends an output $y$ at time $t$ and there is $x \in past_S$, such that $\lambda_S(x) = \lambda_{\mathtt{J}}(y)$ and $\left( \theta_S(x) + \Delta t^+(x) \right) = t$ (thus, $\theta_{\mathtt{J}}(y)$ is a boundary point of $\delta_S(x)$), calling *onSpecTimeout* before *onImplOutput* would lead to the undesirable failure. If there are two specification outputs $x$ and $x'$, such that $\theta_S(x) = \theta_S(x')$ and $x \prec x'$, calling *onSpecOutput*[$x$] should precede calling *onSpecOutput*[$x'$]. The initialization action (*onInitialize*) comes first, while the finalization action (*onFinalize*) is the last action within a time slot. The order between the timeout actions as the order between the output reception actions is insufficient and may be arbitrary. The sequence for checking guards and activating actions within a time slot $t$ is as follows:

1. initialization (*onInitialize*);

2. output reception (*onSpecOutput*[$x$] and *onImplOutput*[$y$], $x \in S[t]$ and $y \in I[t]$);

3. timeouts (*onSpecTimeout*[$x$] and *onImplTimeout*[$y$], $x \in past_S$ and $y \in past_I$);

4. finalization (*onFinalize*).

Note that when we say that some property $\varphi$ holds at time $t$, we mean that $\varphi$ holds after all of the actions activated at time $t$ have completed. For a multi-port system, the monitor can be decomposed into a number of loosely connected sub-monitors serving individual ports. If the specification abstracts away from the inter-port dependencies, the sub-monitors are fully independent and can work in parallel.

**Statement 1 (Monitor correctness)** *An input sequence $w$ is a counterexample for $\mathcal{I}$ being conforming to $\mathcal{S}$ iff the monitor terminates with verdict $= false$.* □

Rigorously speaking, the termination condition $\big(\mathsf{end}(I) + \Delta T^-\big) \leq t$ cannot be checked for "black-box" implementations (a monitor is not able to identify whether the implementation is *quiescent* or *active*). However, for some types of systems (in particular, systems with *covergent behavior*) the condition can be approximated with a checkable one.

**Definition 9 (Convergent behavior)** *The behavior $\mathcal{B} : \mathbb{I} \times \mathbb{T} \to \mathbb{O}$ is called convergent iff the following conditions are met:*

- *for every finite $w \in \mathbb{I}$, there exists $T(w) \in \mathbb{T}$, called the stabilization time, such that for any $t \geq T(w)$, $\mathcal{B}(w,t) = \mathcal{B}\big(w,T(w)\big)$ ($\mathcal{B}(w)$ denotes $\mathcal{B}\big(w,T(w)\big)$);*

- *for every $t \in \mathbb{T}$, $\mathcal{B}(\varepsilon,t) = \varepsilon$ holds (the initial state is quiescent);*

- *for every finite $w,v \in \mathbb{I}$, such that $v \neq \varepsilon$ and $t_0 = \mathsf{begin}(v) > T(w)$, $t \geq t_0$ and $\Delta t \in \mathbb{T}$,*

$$\begin{cases} \mathcal{B}\big(w(v+\Delta t), t+\Delta t\big)_{[t_0+\Delta t, t+\Delta t]} = \mathcal{B}(wv,t)_{[t_0,t]} + \Delta t, \\ t_0 \leq \mathsf{begin}(\mathcal{B}(wv)_{[T(w),\infty)}), \ if \ \mathcal{B}(\cdot)_{[\cdot]} \neq \varepsilon; \end{cases}$$

*where $w + \Delta t$ denotes the sequence constructed from $w$ by adding $\Delta t$ to each time stamp of $w$ (quiescent states are stable).* □

Assuming that the implementation under verification is convergent, the termination condition may be expressed as follows:

$$\Big(T(w) \leq t\Big) \wedge \Big(\big(\mathsf{end}(I_{[0,T(w)]}) + \Delta T^-\big) \leq t\Big).$$

## 3.3   Specifications with Optional Outputs

There are systems where operations in some situations terminate other operations, conflicting with them and of a lower priority. For example, a write operation can be cancelled by another write operation targeted at the same location and started right after the previous one. Due to abstractness, a specification is not able to express precisely under what conditions operations are cancelled and their output is not sent outside. Taking into account such problems, the definition of the specification behavior should be extended. Assume there is an unary relation $\Diamond \subseteq V_S$ marking cancellable outputs (the complement of $\Diamond$ is denoted by $\Box$): if $\Diamond x$, then the output is *optional* (it might be cancelled, but the cancellation condition is unknown or inexpressible in specification terms); if $\Box x$, then the output is *obligatory* (it cannot be cancelled). Note that if some action is cancelled, then all dependent actions are cancelled either.

**Definition 10 (Conformance relation for specifications with optional outputs)** *The implementation behavior $\mathcal{I}$ is said to be conforming to the specification behavior with optional outputs $\mathcal{S}$ iff $\mathbf{dom}\mathcal{I} = \mathbf{dom}\mathcal{S}$ and for all $w \in \mathbf{dom}\mathcal{S}$ and $t \in \mathbb{T}$, there is a relation $\mathcal{M}(w,t) \subseteq \{(x,y) \in \mathsf{past}_S(w,t) \times \mathsf{past}_\mathcal{I}(w,t) \mid \mathsf{match}(x,y)\}$, such that:*

1. $\mathcal{M}(w,t)$ *is a one-to-one relation;*

2. *for each* $x \in \mathsf{past}_{\mathbb{S}}^{\Delta t}(w,t)$,
   - *if* $\square x$, *then there is* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$;
   - *if* $\lozenge x$, *then either there is* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$, *or for each* $x' \in \mathsf{past}_{\mathbb{S}}(w,t)$, *if* $x \preceq x'$, *then there is no* $y \in \mathsf{past}_{\mathbb{J}}(w,t)$, *such that* $(x',y) \in \mathcal{M}(w,t)$.

3. *for each* $y \in \mathsf{past}_{\mathbb{J}}^{\Delta t}(w,t)$, *there is* $x \in \mathsf{past}_{\mathbb{S}}(w,t)$, *such that* $(x,y) \in \mathcal{M}(w,t)$;

4. *for all* $(x,y),(x',y') \in \mathcal{M}(w,t)$, *if* $x \prec x'$, *then* $\theta_{\mathbb{J}}(y) \leq \theta_{\mathbb{J}}(y')$. $\square$

Checking conformance to specifications with optional outputs can be done with a few modifications of the monitor described above. In *onSpecTimeout*, it should be checked whether an event $x$ is optional (the action fails only if $x$ is obligatory). The most difficult part is to track that all events dependent on the cancelled one are also cancelled. Assume that there is $\Delta T_{dep} \in \mathbb{T}$, such that for all $x,x' \in V_{\mathbb{S}}$, if $|\theta_{\mathbb{S}}(x) - \theta_{\mathbb{S}}(x')| > \Delta T_{dep}$, then $x \perp x'$. To describe the monitor, let us introduce a predicate $cancelled_S(x) = (\exists x' \in terms_S . x' \preceq x)$ and a modified version of the primary arbiter: $arbiter_S(X) = \min_{\preceq}(X \setminus terms_S)$.

---

**Action 7** $onSpecOutput[x]$, $x \in S[t]$

**Guard:** *true*
**Input:** $x$
 $past_S \Leftarrow past_S \cup \{x\}$
 **if** $cancelled(x)$ **then**
  $terms_S \Leftarrow terms_S \cup \{x\}$
 **else**
  ...
 **end if**

---

**Action 8** $onSpecTimeout[x]$, $x \in (past_S \setminus terms_S)$

**Guard:** $(\theta_{\mathbb{S}}(x) + \Delta t^+(x)) \leq t$
**Input:** $x$
 **if** $\square x$ **then**
  ...
 **else**
  $terms_S \Leftarrow terms_S \cup \{x\}$
 **end if**

---

**Action 9** $onTermTimeout[x]$, $x \in terms_S$

**Guard:** $(\theta_{\mathbb{S}}(x) + \Delta T_{dep}) \leq t$
**Input:** $x$
 $past_S \Leftarrow past_S \setminus \{x\}$
 $terms_S \Leftarrow terms_S \setminus \{x\}$

---

**Action 10** $onInitialize$

**Guard:** $t = 0$
**Input:** $\varnothing$
 $terms_S \Leftarrow \varnothing$
 ...

---

# 4 Tool Support and Experience

The proposed approach to runtime verification has been implemented in a C++ library named *C++TESK Testing ToolKit* [1]. The library provides users with classes and macros for automated development of test system components, including reference models, monitors (test oracles), stimuli generators, coverage trackers, etc. C++TESK supports testing and monitoring of both hardware and software systems but has been mainly used for hardware designs (namely, for simulation-based verification of microprocessor units). Note that hardware is usually developed in *hardware description languages* (*HDLs*), like Verilog and VHDL, and can be executed (simulated) in a special environment, called *HDL simulator*. The C++TESK facilities for developing reference models of hardware designs (and, consequently, runtime monitors) include means for sending and receiving data packages, forking and joining concurrent threads, modeling time delays and specifying order between data packages. Some of the primitives (the most important within the scope of the paper) are as follows (the syntax here differs from the original one, used in the toolkit):

- `delay(n)` — models a time delay (as an observable outcome, it increments the current time value by `n` time units);

- `recv(in):pkg` — waits until an input package is received at a given input port (`in`); then, returns that package (`pkg`);

- `send(out, pkg, opt)` — sends an output package (`pkg`) via a given output port (`out`) specifying whether the package is obligatory or optional (`opt`);
  (Note that every time a package is sent outside, it is tagged with time interval $[t - \Delta t^-, t + \Delta t^+]$, where $t$ is the sending start time and $\Delta t^\pm$ are user-defined parameters of the transmission port.)

- `depends(pkg1, pkg2)` — states that an output package (`pkg1`) depends on or causally related to some other package (`pkg2`), input or output.
  (This probably answers the question raised in the beginning of Section 3 of where a specification trace, namely partial ordering of its events, is taken from.)

Differences in hardware complexity, verification purposes and amounts of resources lead to the variety of model types and model abstraction levels. Abstraction is a well-known way for fighting complexity and facilitating model development. Though the verification quality is likely to be lower in case of simpler reference models, if there is a strict deadline (and it is often so), there is no other way out. Event ordering and timing are the main subjects for abstraction in hardware designs and other concurrent time-dependent systems. We use the following classification of the reference models according to the time modeling accuracy: (1) *untimed models* (represent only general information on the cause-effect relation of their inputs and outputs, while the timing is not modelled at all: $\Delta t^\pm = \infty$), (2) *time-approximate models* (contain the detailed specification of the event ordering, including some internal arbitration schemes, but the timing is approximate: $\Delta t^\pm \leq T$, where $T$ has a value of several tens of time units) and (3) *time-accurate models* (implement the exact, or almost exact, event ordering and timing: $\Delta t^\pm \leq 1$).

The proposed methodology has been used for verification of a number of units of different industrial microprocessors. Our experience was originally presented in [9], and since then we have verified a table lookup unit, an l2-cache bank controller and an instruction buffer. Also, testbenches and monitors for several previously tested components (a north bridge data switch and a memory access unit) required improvement according to the modifications of the units. The newest information of the approach application is shown in Table 1. As it can be seen from the table, the methodology supports runtime verification by means of abstract models (being available at early design stages) and, at the same time, by means of up to time-accurate models (being available typically at finishing design stages). Moreover, the approach allows reusing reference models across the hardware development cycle, which is really important in the industrial settings.

The first version of C++TESK supported only accurate reference models (it was required that a model knows the exact ordering of events on each of the output ports). Having received feedback from C++TESK users (everyone is welcome to join the community), the toolkit has been modified. Mostly, it concerns a problem of lack of unit-level specifications even for almost finished hardware designs. It is impossible to create an accurate model without detailed knowledge of the unit functionality and timing. Regular interviewing of engineers takes a lot of time and is inconvenient. Two major solutions of the problem have been proposed besides forcing the developers to write the specifications. The first solution is to reuse parts of a more complicated system-level model (emulating behavior of the whole microprocessor). Though such parts are rather abstract (as a rule, system-level models are developed in an untimed manner), they are really useful for early-stage verification. The second solution is to develop approximate reference models by means of C++TESK and to refine them if necessary.

| Microprocessor Unit | Development Stage | Model Abstraction Level | |
|---|---|---|---|
| | | From | To |
| Translation lookaside buffer | Late / finishing | Time-approximate model | Time-accurate model |
| Floating point unit | Late / finishing | Untimed model | — |
| Non-blocking L2-cache | Middle / late | Time-approximate model | — |
| North bridge data switch | Middle / late / finishing | Time-approximate model | Time-accurate model |
| Memory access unit | Early / middle | Untimed model | Time-accurate model |
| System interrupt controller | Early / middle | Untimed model | Time-approximate model |
| Table lookup unit | Late | Time-approximate model | — |
| L2-cache bank controller | Late | Time-accurate model | — |
| Instruction buffer | Late / finishing | Time-accurate model | — |

Table 1: Experience of the approach application

# 5   Related Work

There are several works on model-based testing and monitoring that have similarities with our approach. Some of them are mentioned below.

In [7], a *partial order input/output automaton* (*POIOA*), where each transition is associated with an almost arbitrary ordered set of inputs and outputs, is used to represent the expected behavior. The key idea is to obtain two POIOAs (representing behavior of specification and implementation) and to check their conformance. There is a way to derive a test suite that guarantees fault detection defined by a POIOA-specific fault model: *missing output faults*, *unspecified output faults*, *weaker precondition faults*, *stronger precondition faults* and *transfer faults*. If the following assumptions are satisfied: an unspecified input is detectable, specified ordering of outputs can be observed, response time is bounded, and each specification transition can be modeled as a single implementation transition, then it is possible to set up conformance between two POIOAs. Comforming implementation accepts any input compatible with the specification (and may accept more) and produces outputs defined by the specification in an order compatible with the specification. If the POIOAs are not conforming, it is considered as wrong behavior of the implementation according to the fault model. The main difficulty in the approach, in our opinion, is to represent behavior of specification and implementation by the proposed formalism.

In [3], the approach to passive testing based on *invariants* is presented. Invariants are used as a means of representing the most relevant expected properties of the implementation, which should be exhibited in response to the corresponding test sequences. Two types of invariants are of usage: (1) *timed consequent invariants* and (2) *timed observational invariants*. The first type is used to check that an event happens (within certain time bounds) after a given trace of events. The second type is used to check that a given sequence of events always occur (within certain time bounds) between two given events. The correctness of the implementation behavior is verified in two steps. The first step is to check the correctness of the invariants with respect to a given specification. The second step is to check the correctness of a trace, recorded from the implementation, with respect to the invariants. We think, that this approach is applicable to monitoring of complex timed systems, but it is not clear how to maintain the sets of invariants (which might be huge) during the system life cycle.

The approach proposed in [13] allows usage of implicitly defined *asynchronous finite state machines* (*AFSMs*) for model-based testing of complex distributed systems. The implementation behavior is verified only in *quiescent states* of the FSM model. Thus, it is required that there is a predicate identifying such kind of states. The testing step is done as follows. First, all outputs are collected and their partial

order is determined. Then, all possible linearizations of the events are enumerated and checked. If all of them fail (with respect to the specification), then the implementation is not conforming to the specification. As checking is performed in quiescent states only, the approach is hardly applicable to runtime monitoring (where there may be arbitrary input sequences, and such states are rarely visited).

## 6  Conclusion

On-the-fly analysis of system behavior is an integral part of dynamic verification of software and hardware systems. A lot of formalisms have been proposed to express correctness properties for systems of different types, and a great number of methods have been suggested to check whether system executions are conforming to the specified properties. None of them is perfect, we think, but all together they cover a vast spectrum of verification and monitoring tasks. Among the variety of specification approaches, executable models, written in high-level programming languages, have a significant niche. First of all, such models are rather universal and allow expressing a broad range of behavioral and structural properties. Besides, programming languages (especially general-purpose languages, like C and C++) are widely spread in the engineering community.

Our work focuses on using executable models for runtime verification of reactive systems, including, in particular, time-dependent systems. The problem is not as simple as it looks at first sight. The naive checking that a system and its model produce the same outputs at the same time is inadequate in the majority of cases. The model may abstract away from many features implemented in the system under verification such as event ordering and accurate timing (at least it should be abstracted from the implementation bugs). We suppose that conformance relations used for runtime verification can be configured in several ways: (1) by introducing an independency relation over the model events, (2) by extending time points of the model outputs to time intervals and, finally, (3) by marking some of the model outputs as being optional.

Basing on this idea, we have developed a method for system monitoring and proved its correctness. The formalization is based on the theory of traces and partially ordered multisets. The method has been implemented in C++TESK, an open-source toolkit for hardware modeling, analysis and verification, and has been successfully used in about 10 projects on simulation-based verification of microprocessor units. Our future research is aiming at failure diagnostics, which is a deeper analysis of specification and implementation traces being carried out offline. The goal is to explain what in particular went wrong during the monitoring and give a hint to developers where the bugs are localized.

## 7  Bibliography

## References

[1] *C++TESK Homepage*. Available at `http://forge.ispras.ru/projects/cpptesk-toolkit/`.

[2] R. Alur & D.L. Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[3] C. Andrés, M.G. Merayo & M. Núñez (2012): *Formal Passive Testing of Timed Systems: Theory and Tools*. *Software Testing, Verification & Reliability* 22(6), pp. 365–405, doi:10.1002/stvr.1464.

[4] H. Barringer, D. Rydeheard & K. Havelund (2007): *Rule Systems for Run-Time Monitoring: From Eagle to RuleR*. In: *Proceedings of 7th International Workshop on Runtime Verification. Revised Selected Papers*, pp. 111–125, doi:10.1007/978-3-540-77395-5_10.

[5] A. Bauer, M. Leucker & C. Schallhart (2011): *Runtime Verification for LTL and TLTL*. ACM Transactions on Software Engineering and Methodology 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800.

[6] B. Bloom & M. Kwiatkowska (1991): *Trade-offs in True Concurrency: Pomsets and Mazurkiewicz Traces*. Technical Report TR 91-1223, Cornell University.

[7] G. von Bochmann, S.Haar, C.Jard & G.-V. Jourdan (2008): *Testing Systems Specified as Partial Order Input/Output Automata*. In: *Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, Springer-Verlag, Berlin, Heidelberg, pp. 169–183, doi:10.1007/978-3-540-68524-1-13.

[8] D.V. Chieu & D.V. Hung (2012): *Timed Traces and Their Applications in Specification and Verification of Distributed Real-time Systems*. In: *Proceedings of the Third Symposium on Information and Communication Technology*, pp. 31–40, doi:10.1145/2350716.2350723.

[9] M. Chupilko & A. Kamkin (2011): *A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels*. In: *Proceedings of the 12th Latin-American Test Workshop*, pp. 1–6, doi:10.1109/LATW.2011.5985902.

[10] E.M. Clarke, O. Grumberg & D.A. Peled (1999): *Model Checking*. The MIT Press.

[11] R.M. Hierons, K. Bogdanov, J.P.Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A.J.H. Simons, S. Vilkomir, M.R. Woodward & H. Zedan (2009): *Using Formal Specifications to Support Testing*. ACM Computing Surveys 41(2), pp. 9:1–9:76, doi:10.1145/1459352.1459354.

[12] V.P. Ivannikov, A.S. Kamkin, A.S. Kossatchev, V.V. Kuliamin & A.K. Petrenko (2007): *The Use of Contract Specifications for Representing Requirements and for Functional Testing of Hardware Models*. Programming and Computer Software 33(5), pp. 272–282, doi:10.1134/S0361768807050039.

[13] V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev & I. Bourdonov (2003): *Integration of Functional and Timed Testing of Real-Time and Concurrent Systems*. In M. Broy & A. Zamulin, editors: *Perspectives of System Informatics*, Lecture Notes in Computer Science 2890, Springer Berlin Heidelberg, pp. 450–461, doi:10.1007/978-3-540-39866-0-45.

[14] W.K. Lam (2005): *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Prentice Hall.

[15] J. Laski & W. Stanley (2009): *Software Verification and Analysis: An Integrated, Hands-On Approach*. Springer.

[16] M. Leucker (2000): *On Model Checking Synchronised Hardware Circuits*. In: *Proceedings of the 6th Asian Computing Science Conference*, Lecture Notes in Computer Science 1961, Springer, pp. 182–198, doi:10.1007/3-540-44464-5_14.

[17] G. Luo, R. Dssouli, G. von Bochmann, P. Venkataram & A. Ghedamsi (1993): *Generating Synchronizable Test Sequences Based On Finite State Machine with Distributed Ports*. In: *Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems*, pp. 53–68.

[18] A. Mazurkiewicz (1987): *Trace Theory*. In: *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, Springer-Verlag New York, Inc., New York, NY, USA, pp. 279–324, doi:10.1007/3-540-17906-2_30.

[19] B. Patel (2010): *A Monitor-Based Approach to Verification*. EE Times.

[20] V.R. Pratt (1984): *The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial*. In: *Seminar on Concurrency*, pp. 180–196, doi:10.1007/3-540-15670-4_9.

[21] K. Sen & G. Rosu (2003): *Generating Optimal Monitors for Extended Regular Expressions*. Electronic Notes in Theoretical Computer Science 89(2), pp. 162–181, doi:10.1016/S1571-0661(04)81051-X.

[22] B. Wile, J. Goss & W. Roesner (2005): *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann.