Real-Time Syst (2007) 36: 159–198
DOI 10.1007/s11241-007-9013-6

Predictable real-time software synthesis

Jinfeng Huang · Jeroen Voeten · Henk Corporaal

Published online: 28 March 2007

Abstract Formal theories for real-time systems (such as timed process algebra, timed automata and timed petri nets) have gained great success in the modeling of concurrent timing behavior and in the analysis of real-time properties. However, due to the ineliminable timing differences between a model and its realization, synthesizing a software realization from a model in a predictable way is still a challenging research topic. In this article, we tackle this problem by solving a set of sub-problems. The solution is based on the theoretical results for property prediction proposed in Huang et al. (2003, Real-time property preservation in approximations of timed systems. In: Proceedings of 1st ACM and IEEE international conference on formal methods and models for codesign. IEEE Computer Society, Los Alamitos, pp 163–171) and Huang (2005, Predictability in real-time system design. PhD thesis, Eindhoven University of Technology, The Netherlands), where quantitative property relations are established between two absolute/relative "close" real-time systems. This theory basically implies that if two systems are "close", they enjoy "similar" properties. These results cannot be directly applied in practice though, because a model and its realization typically have infinitely large absolute and relative timing differences. We show that this infinite time gap can be bridged through a sequence of carefully constructed intermediate time domains. Then the property-prediction results can be applied to any pair of adjacent time domains in the sequence. Consequently, real-time properties of the implementation can be predicted from the model. We propose two parameterized hypotheses to characterize the timing differences in the sequence and to guide a correctness-preserving design process. It is shown that these hypotheses can be incorporated in a concrete tool set. We demonstrate the feasibility of the predictable synthesis approach through the design of a railroad crossing system.

J. Huang (✉) · J. Voeten · H. Corporaal
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: j.huang@tue.nl; jinfeng.huang@gmail.com

J. Voeten
Embedded Systems Institute, Eindhoven, The Netherlands

Keywords Real-time systems · Formal methods · Program synthesis

1 Introduction

In the past decade, many formal theories to construct models for real-time systems have been proposed in academic fields. Examples include timed automata (Alur and Dill 1994), timed process algebra ATP (Nicollin and Sifakis 1994) and timed petri nets (Stotts and Pratt 1985), just to mention a few. These formalisms have gained great success in the analysis of real-time systems at a high level of abstraction. To apply these formal theories in industrial environments, modeling languages based on formal theories have been proposed. These languages possess the merits provided by formal theories. Furthermore, they provide more facilities (such as data expressions) to describe real-time systems in detail and bring models closer to realizations. Examples of these modeling languages are timed automata extended with data types and tasks (Larsen et al. 1997; Amnell et al. 2003), SDL-2000 based on distributed real-time ASM (Glasser et al. 2003), Ptolemy (in the CSP domain) based on timed CSP (Smyth 1998), and POOSL (van der Putten and Voeten 1997) based on timed CCS.

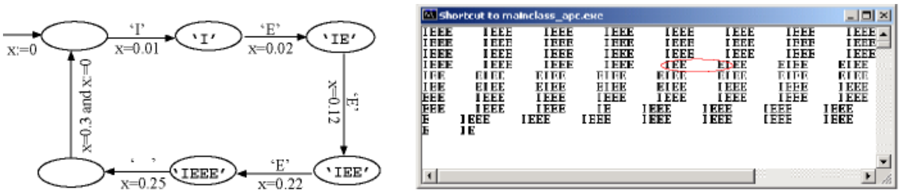
In the timing semantics of these formal theories and modeling languages, actions are often assumed to be instantaneous. This assumption brings many benefits to the modeling and the analysis of real-time systems:

- Concurrent processes do not compete for time resources with each other, since the execution of actions does not consume time. Therefore the timing behavior of each process is left unchanged when different processes are integrated.
- Monitoring (or analysis) code does not consume execution time. Hence, the timing behavior of the monitoring code does not interfere with the timing behavior of the monitored system.
- Adding design details to or removing design details from a part of a system has no influence on the computation time of the other parts of the system. This reduces system design complexity, since abstractions or refinements can be performed locally.
- The timing behavior of the system is not affected by techniques such as caches and pipelines adopted by the underlying platform.

1.1 Problem illustration

Although the assumption that actions are instantaneous offers many benefits for modeling and analyzing real-time systems, a smooth path is still lacking to generate the realization from its model while preserving the desired properties. This is demonstrated by the following example.

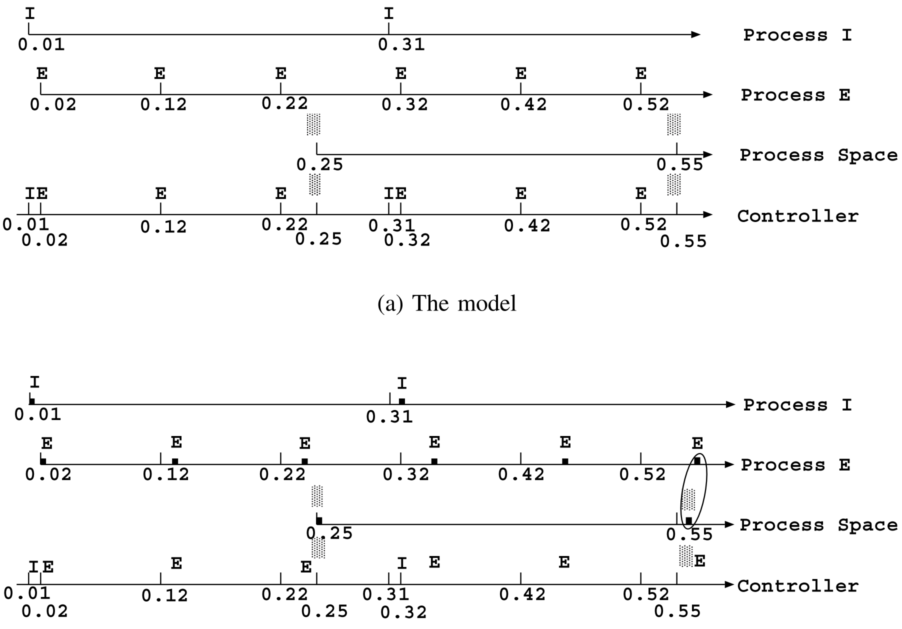
Example 1 Consider a controller for a flash board showing 4 consecutive letters ‘*IEEE*’. A possible design solution to the controller is to use three parallel processes **I**, **E** and **S**. **I** emits letter I every 0.3 seconds, **E** emits letter E every 0.1 seconds and **S** issues four blank spaces every 0.3 seconds to erase the letters. The three processes start from 0.01, 0.02 and 0.25 seconds respectively.



```
IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE
IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE
IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE
IEEE    IEEE    IEEE    IEEE    IEE     EIEE    EIEE    EIEE
IEE    EIEE    EIEE    EIEE    EIEE    EIEE    EIEE    EIEE
IEE    EIEE    EIEE    EIEE    EIEE    IEEE    IEEE    IEEE
EEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE
EEE    IEEE    IEEE    IE    IEEE    IEEE    IEEE    IEEE
E    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE    IEEE
E    IE
```

(a) The model (b) The output of a realization

Fig. 1 The IEEE flashboard controller example



(a) The model

(b) The realization

Fig. 2 The difference between timing semantics

We made a model of the system using the SDL-2000 language in the TAU G2 tool from Telelogic, where actions are performed instantaneous (Glasser 1998). A timed state-transition diagram of the model is given in Fig. 1(a), where x is a clock used to express timing constraints on actions (or events) and $x := 0$ denotes that x is reset to 0. Based on the timing semantics of the model (exemplified in Fig. 2(a)), it is easy to see that the model performs correctly. Figure 1(b) gives the output of a realization automatically generated by TAU G2. The figure shows that after some execution time, a wrong output sequence is observed. The inconsistency between the model and the realization can be attributed to the difference between the timing semantics of the modeling language and that of the implementation language. In the model timing is based on a "virtual" notion of time, where the execution of actions take no "virtual time”. This “virtual time” concept coincides with the notion of global time in many formal frameworks (such as Alur and Dill 1994 and Henzinger et al. 1992) which assign time to states or actions in a trace. In the realization, each action takes some amount of “physical time” to execute and timing expression “*delay t*” should be interpreted as “*at least t seconds*” due to the scheduling overhead and the execution time consumed by other tasks. A comparison of the two different timing semantics (i.e. in the model and in the realization) is illustrated in Fig. 2. We can see that durational actions can result in accumulated timing errors for each single process, changing the relative timing relations between concurrent real-time processes in Fig. 2(b).

The example illustrates that unexpected behaviors can be observed in the realization. The primary reason for this is that the semantics of the realization does not respect the semantics of the model. Consequently one cannot reliably draw conclusions about the realization, based on the analysis of the model.

1.2 Solution sketch

To solve the problem illustrated in the previous subsection, we need a systematic way to deal with timing differences between the model and the realization. In particular we would like to generate the realization on the basis of the timing semantics of the model. More specifically, both the model and the realization are considered to be sets of traces, each of which represents an execution. In our approach any trace of the realization is generated from a trace in the model implying that they have the same state (or action) sequence. Furthermore, the times of occurrences of corresponding states (or actions) in both traces are synchronized. Consequently, the qualitative properties (such as safety properties) of the realization are the same as those of the model and the quantitative properties (such as deadline properties) of the realization are “close” to those of the model.

Since timeliness is an essential feature of real-time systems, it is important to quantify the “closeness” between the quantitative properties of a model and its realization, so that we can predict whether the realization is correct. In (Huang 2005), we define absolute and relative timing proximity between traces, on which property relations between traces are quantitatively established.[1] However, these results cannot be applied directly in a synthesis approach, due to the following facts.

- The absolute and relative timing proximities are defined on the basis of the observation times of states (or actions). It is often problematic to measure them in practice.
- The model (in the virtual time domain) and the realization (in the physical time domain) often have infinitely large absolute and relative timing proximities. Note that the concept of physical time is more complex than we have explained thus far. The physical time concept used here denotes the time of the system environment, whereas physical time synchronized with the virtual and physical time, as explained before, refers to the time of a hardware clock. A detailed discussion is found in Sect. 4.

[1] In (Huang et al. 2003) we proved a special case based on absolute timing differences. The general results based on both absolute timing differences and relative timing differences can be found in (Huang 2005).

To address these issues, we introduce the concept of clocks to characterize time domains. An untimed trace observed in different time domains can result in different timed traces. Then the absolute and relative timing proximities between these traces can be estimated by the clock deviation or drift, which can be known in practice. Based on the clock concept, we bridge the timing gap between the model and the realization by introducing a number of intermediate time domains. Between each pair of adjacent time domains, we use the results of (Huang 2005) for property prediction.

In short, the synthesis approach proposed in this article has the following features.

- A unified formal model is used to analyze both quantitative and qualitative properties.
- By construction, the synthesis approach ensures that the realization has the same qualitative properties as the model.
- The synthesis approach supports prediction of quantitative properties of the realization based on those of the model.
- The synthesis approach has been incorporated in a tool set (POOSL + Rotalumis) demonstrating the feasibility of the approach.

1.3 Organization

The remaining part of the article is structured as follows.

In Sect. 2, we summarize the theoretical results proven in (Huang 2005) for predicting real-time system properties. More specifically, we consider a real-time system to be a set of timed state sequences each of which represents an execution. For any two timed state sequences with the same sequence of observable states but with different time durations of corresponding states, properties of one sequence can be predicted from those of the other based on their absolute or relative timing proximities.

In our synthesis approach, the same state (or action) sequence is observed in different time domains resulting in different timed state sequences. For instance, a sequence of states (or actions) is observed in the virtual time domain in the model and is observed in the physical time domain in the realization. To know the distance between these timed state sequences, we need to know the time duration of each state in the sequence, which is not always possible in practice. In Sect. 3, we define the concept of clocks to specify time domains. We show that the proximity of corresponding timed state sequences observed in different time domains can be estimated on the basis of clock deviation or drift. Furthermore, we prove in Sect. 3 that the infinitely large absolute and relative distances between two time domains can be split into several bounded absolute and relative distances by introducing proper intermediate time domains.

In Sect. 4, we investigate the concrete time domains involved in real-time software synthesis. As illustrated in Fig. 3, we assume that the model operates in the virtual time domain, the realization is observed in a digital hardware time domain driven by the clock of the target platform, and that the realization is also observed in a reference time domain, in which it interacts with the environment. By using the concepts of the previous section, an auxiliary time domain is introduced between the digital and reference time domains to bridge their (infinite) timing gap. Since the distances between these time domains have a direct impact on the property prediction of the realization,

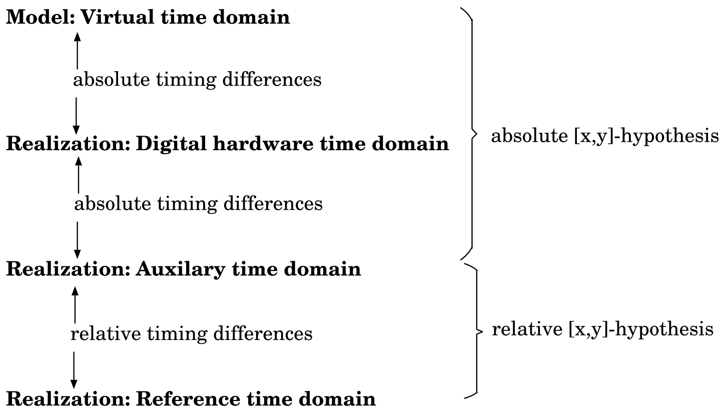

Model: Virtual time domain
absolute timing differences
Realization: Digital hardware time domain
absolute timing differences
Realization: Auxilary time domain
relative timing differences
Realization: Reference time domain
absolute [x,y]-hypothesis
relative [x,y]-hypothesis

Fig. 3 Time domains in real-time system design

we also discuss how to construct an "optimal" auxiliary time domain such that the prediction results are tight.

In Sect. 5, we summarize the timing relations discussed in the previous section into two parameterized hypotheses. The absolute hypothesis assumes that the absolute timing differences between the model and the realization in the auxiliary time domain are bounded. The parameters of the absolute hypothesis capture the bounds of the absolute timing differences. The absolute timing differences consist of two components as shown in Fig. 3. The relative hypothesis assumes that the realization in the reference time domain has bounded relative timing differences to that in the auxiliary time domain (see Fig. 3). The parameters of the relative hypothesis capture the bounds of relative timing differences.

In Sect. 6, we demonstrate that the two hypotheses can indeed be incorporated into a software synthesis tool set. The POOSL language which semantics is based on an extension of timed CCS is used to model concurrent real-time systems. The Rotalumis tool transforms a POOSL model to its realization such that both the model and the realization have the same observable behavior. Therefore, the realization has the same qualitative real-time properties as the model by construction. Furthermore, by estimating the parameters of both hypotheses, we can predict the quantitative real-time properties of the realization and ensure its correctness.

Section 7 applies the proposed synthesis approach to design a railroad crossing system. Sections 8, 9 and 10 discuss related work, open issues and conclusions respectively.

2 Property relations between real-time systems

In this section, we briefly summarize the theoretical results proven in (Huang 2005), the intuition of which is sketched in Fig. 4. In this figure, real-time system S_2 is "close" to S_1. If it can be verified that S_1 satisfies a (quantitative or qualitative) real-time property, then a "corresponding" property satisfied by S_2 can be predicted. In (Huang 2005), the figure is completed based on two different proximity measures between real-time systems.

Fig. 4 Real-time property relations between two neighboring real-time systems

$$\begin{tikzcd} S_1 \arrow[r, "satisfies"] & P \arrow[d, "?"] \\ S_2 \arrow[u, "close"] \arrow[r, "satisfies"] & ? \end{tikzcd}$$

2.1 Measuring proximity between real-time systems

A real-time system can be considered to be a set of traces, each of which represents an execution of the system. Each trace (called *timed state sequence*) consists of a sequence of duration states (δ_i, I_i), where δ_i is the state observed during non-empty time interval I_i. For example, a timed state sequence of the 'IEEE' flashboard controller in Example 1 is:

$$(\emptyset, [0, 2.01)), \quad (\text{'I'}, [2.01, 2.02)), \quad (\text{'IE'}, [2.02, 2.12)), \quad \ldots.$$

The length of I_i (denoted as $|I_i|$) is called the *time-duration* of state δ_i. The left-end of I_i (denoted as $l(I_i)$) is called the *time-stamp* of the state transition from δ_{i-1} to δ_i. Alternatively, a timed state sequence can be viewed as a pair of sequences: a state sequence $\bar{\delta}$ formed by states δ_i, and a time interval sequence $\bar{I}$ formed by time intervals I_i. It is easy to see that the length (number of states) of $\bar{\delta}$ is identical to the length of $\bar{I}$, which is denoted as $n(\bar{I})$.[2]

Proximity function D^s_a is defined to measure the absolute difference between two timed state sequences based on their corresponding time stamps.

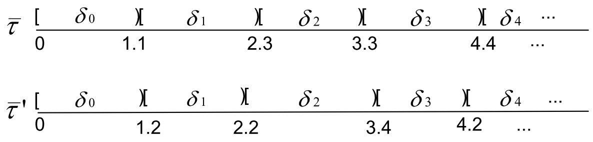
Definition 1 For any pair of timed state sequences $\bar{\tau}$ and $\bar{\tau}'$ with the same state sequence, $D^s_a(\bar{\tau}, \bar{\tau}') = [x, y]$, where $x = \inf\{l(I'_i) - l(I_i) \mid i < n(\bar{I})\}$ and $y = \sup\{l(I'_i) - l(I_i) \mid i < n(\bar{I})\}$.

$D^s_a(\bar{\tau}, \bar{\tau}')$ reflects that the state transitions in $\bar{\tau}'$ are at least x seconds and at most y seconds later than their corresponding transitions in $\bar{\tau}$. If $y < 0$, the transitions in $\bar{\tau}'$ are always earlier than their corresponding transitions in $\bar{\tau}$. It is worth noticing that proximity function D^s_a is directional. That is, $\mathcal{D}^s_a(\bar{\tau}, \bar{\tau}')$ is typically not equal to $\mathcal{D}^s_a(\bar{\tau}', \bar{\tau})$. However, it is easy to prove that $\mathcal{D}^s_a(\bar{\tau}, \bar{\tau}') = [x, y]$ implies $\mathcal{D}^s_a(\bar{\tau}', \bar{\tau}) = [-y, -x]$.

Example 2 The two timed state sequences $\bar{\tau}$ and $\bar{\tau}'$ shown in Fig. 5 have the same state sequence with five state transitions. It is easy to calculate that $\sup\{l(I'_i) - l(I_i) \mid i < 5\} = 0.1$ and $\inf\{l(I'_i) - l(I_i) \mid i < 5\} = -0.2$. Therefore, $\mathcal{D}^s_a(\bar{\tau}, \bar{\tau}') = [-0.2, 0.1]$.

Since a real-time system is seen as a set of timed state sequences, the proximity between real-time systems is defined on the basis of the proximity between their timed state sequences. Definition 2 gives a proximity measure based on absolute timing differences.

[2] $n(\bar{I})$ can be finite or countable infinite.



$\bar{\tau}$
δ_0 δ_1 δ_2 δ_3 δ_4 ...
0 1.1 2.3 3.3 4.4 ...
$\bar{\tau}'$
δ_0 δ_1 δ_2 δ_3 δ_4 ...
0 1.2 2.2 3.4 4.2 ...

Fig. 5 Two finite timed state sequences of Example 2

Definition 2 Let S_1 and S_2 be two real-time systems. S_2 is called absolute $[x, y]$-close to S_1 iff for any timed state sequence $\bar{\tau}'$ in S_2, there exists a sequence $\bar{\tau}$ in S_1, such that $D_a^s(\bar{\tau}, \bar{\tau}') \subseteq [x, y]$.

Similar to the absolute case, proximity function D_r^s is defined to measure the *r*elative timing difference between two timed *s*tate sequences.

Definition 3 For any pair of timed state sequences $\bar{\tau}$ and $\bar{\tau}'$ with the same state sequence, $D_r^s(\bar{\tau}, \bar{\tau}') = [x, y]$, where $x = \inf\{\frac{|I_i'|}{|I_i|} \mid i < n(\bar{I})\}$ and $y = \sup\{\frac{|I_i'|}{|I_i|} \mid i < n(\bar{I})\}$. In case that both $|I_i'|$ and $|I_i|$ are zero (or ∞), the value of $\frac{|I_i'|}{|I_i|}$ is left undefined.

In the above definition, $\frac{|I_i'|}{|I_i|}$ gives the ratio between the durations of the i-th state in $\bar{\tau}'$ and $\bar{\tau}$. Assume we use two different clocks to measure the duration of each state δ_i and correspondingly obtain two timed state sequences $\bar{\tau}$ and $\bar{\tau}'$. Then $\frac{|I_i'|}{|I_i|}$ is also the ratio between the average change rates of the clocks during the observation of state δ_i.

Example 3 Consider the two timed state sequences shown in Fig. 5. We can easily calculate that $\sup\{\frac{|I_i'|}{|I_i|} \mid i < 5\} = \frac{6}{5}$, and that $\inf\{\frac{|I_i'|}{|I_i|} \mid i < 5\} = \frac{8}{11}$. Hence, $\mathcal{D}_r^s(\bar{\tau}, \bar{\tau}') = [\frac{8}{11}, \frac{6}{5}]$.

Similar to the absolute case, we can now define a proximity measure between real-time systems for the relative case.

Definition 4 Let S_1 and S_2 be two real-time systems. S_2 is called relative $[x, y]$-close to S_1 iff for any timed state sequence $\bar{\tau}'$ in S_2, there exists a sequence $\bar{\tau}$ in S_1, such that $D_r^s(\bar{\tau}, \bar{\tau}') \subseteq [x, y]$.

2.2 Specifying real-time properties

We choose linear temporal logic *MTL* (Koymans 1990) to express real-time properties. *MTL* formulas are formed by the following syntactic phrases.

$$\varphi ::= p \mid \neg p \text{ “not”} \mid \varphi_1 \vee \varphi_2 \text{ “or”} \mid \varphi_1 \wedge \varphi_2 \text{ “and”} \mid \varphi_1 \mathsf{U}_I \varphi_2 \text{ “until”} \mid \varphi_1 \mathsf{V}_I \varphi_2 \text{ “unless”}.$$

Here p is an “atomic proposition” and time-bound I is an interval of nonnegative reals. If I is $[0, \infty)$, we omit time-bound I of temporal operators.

The *MTL* logic presented here is actually a kind of “negation-free” logic, where the negation is only allowed in front of atomic propositions. However, any conventional *MTL* formula, which has operators like negation and implication, can always

$$\begin{tikzcd} S_1 \arrow[r, "\text{satisfies}"] & P \arrow[d] \\ S_2 \arrow[u, "\text{absolute [x,y]-close}"] \arrow[r, "\text{satisfies}"] & R_a^{(y-x)\Box}(P) \end{tikzcd}$$

(a) Absolute timing difference case

$$\begin{tikzcd} S_1 \arrow[r, "\text{satisfies}"] & P \arrow[d] \\ S_2 \arrow[u, "\text{relative [x,y]-close}"] \arrow[r, "\text{satisfies}"] & R_r^{[x,y]}(P) \end{tikzcd}$$

(b) Relative timing difference case

Fig. 6 Property relations between real-time systems

be converted to a "negation-free" formula. More information can be found in (Huang 2005). Two commonly used operators $\Box\varphi$ ("always") and $\Diamond\varphi$ ("eventually") can also be represented as $\mathit{false}\mathsf{V}\varphi$ and $\mathit{true}\mathsf{U}\varphi$ respectively.

The *MTL* logic can express a wide range of real-time properties. For example, consider a time bounded response property φ, expressing that every stimulus *req* is always followed by a response *ack* within 2 to 6 seconds. φ can be formalized as $\Box(r \to \Diamond_{[2,6]}a)$, where r and a represent atomic propositions "a *req* is observed" and "an *ack* is observed" respectively.

To establish the property relations between real-time systems, functions $R_a^{\epsilon\Box}$ and $R_r^{[x,y]}$ are defined over *MTL* formulas (Huang 2005). Loosely speaking, $R_a^{\epsilon\Box}(\varphi)$ is the formula that extends the time bounds of *until* operators and shrinks the time bounds of *unless* operators in φ by an absolute deviation of ϵ. As an example, consider response property φ defined previously. Then $R_a^{0.2\Box}(\varphi)$ is the real-time property $\Box(r \to \Diamond_{[2-0.2,6+0.2]}a)$, which states that every stimulus *req* is always followed by an *ack* response within 1.8 to 6.2 seconds. Function $R_r^{[x,y]}$ changes the time bounds in a formula with scale factors x and y. For example, $R_r^{[0.8,1.25]}(\varphi)$ is the real-time property $\Box(r \to \Diamond_{[2\times 0.8,6\times 1.25]}a)$ stating that every stimulus *req* is always followed by a response *ack* within 1.6 to 7.5 seconds.

2.3 Property relations between real-time systems

In this section, we present two theorems, which are proven in (Huang 2005). The theorems are visualized in Fig. 6. Both theorems show that real-time properties of one real-time system can be predicted from another based on their (absolute/relative) timing differences. They serve as the theoretical basis to predict real-time properties of the realization from those of the model.

Theorem 1 *Let φ be an MTL formula. Let S_1, S_2 be two real-time systems, such that S_2 is absolute $[x,y]$-close to S_1. If $S_1 \models \varphi$, then $S_2 \models R_a^{(y-x)\Box}(\varphi)$.*

Theorem 2 *Let φ be an MTL formula. Let S_1, S_2 be two real-time systems such that S_2 is relative $[x,y]$-close to S_1. If $S_1 \models \varphi$, then $S_2 \models R_r^{[x,y]}(\varphi)$.*

3 Proximity measures vs. time domains

In the previous section, we defined two proximity functions to measure the absolute and relative proximity between two real-time systems. However, because the time duration of each state during execution is not always possible to obtain, we cannot apply these functions to measure proximities directly. Furthermore, the model and the realization often have unbounded absolute and relative timing differences. For example, two interleaved simultaneous actions (transitions) are observed at the same virtual time point, while they are observed at different physical time points. Consequently, the relative timing differences between the model and the realization are unbounded in this case. Moreover, in practice, absolute timing differences between the model and the realization can be unbounded too (see also Sect. 4).

To overcome these problems and apply the results given in Sect. 2 to real-time software synthesis, we formally define the concept of clocks and demonstrate that the absolute/relative distance between timing behaviors can be estimated by the clock deviation and drift. Furthermore, we show that the infinite deviation and drift between two clocks may be captured by several consecutive bounded deviations and drifts.

In the physical reality, time is often counted by a clock. Assume there exists a true physical time, "which flows equably without relation to anything external" (Newton 1999/1687). Each clock C is used to measure the progress of the true physical time, and can be considered to be a function which maps each time point t in the true physical time domain T to another time domain T_c counted by C. In the following, we define several concepts related to clocks.

Definition 5 *Analog clock*: An analog clock C is a function defined as:

$$C(t) = \int_0^t f(t')\,dt',$$

where $f(t')$ is the change rate of C at true physical time t' ($t' \in R^{\geq 0}$).

Definition 6 *Digital clock*: A Digital clock C is a function defined as:

$$C(t) = i \cdot \Delta, \quad \text{if } t \in \left[\sum_{j=1}^{i} p(j), \sum_{j=1}^{i+1} p(j)\right).$$

Here Δ is the step-width of the clock, which refers to the time elapsed between successive ticks in the digital time domain, and step function $p(i)$ represents the true physical time progress during the i-th step of the clock. For brevity, $s(i)$ abbreviates $\sum_{j=1}^{i} p(j)$ and denotes the time span of the first i steps in the true physical time domain.

Several clock examples, visualized in Fig. 7, are as follows.

- $C_p(t) = t$. C_p is a perfect physical clock, and its change rate is $f(t) = 1$.
- $C_a(t) = \frac{2}{3}t^{\frac{3}{2}}$. C_a is an analog clock, and its change rate is $f(t) = t^{\frac{1}{2}}$.
- $C_d(t) = i$ when $t \in [1.5i, 1.5(i+1))$. C_d is a digital clock. In this example, $\Delta = 1$ and $p(i) = 1.5$ for all i, indicating that the true physical time advances 1.5 seconds when clock C_d advances 1 second.

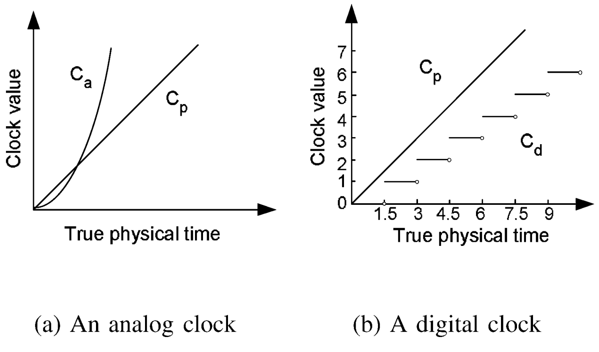


Fig. 7 Several clock examples

Definition 7 *Clock deviation*: For two clocks C_1 and C_2, the deviation of C_2 w.r.t. C_1 at true physical time t is defined by the following function:

$$E^a_{(C_1,C_2)}(t) = C_2(t) - C_1(t).$$

For example, in the previous clock examples, the clock deviation of C_a (and C_d) w.r.t. C_p is calculated by $E^a_{(C_p,C_a)}(t) = C_a(t) - C_p(t) = \frac{2}{3}t^{\frac{3}{2}} - t$ and $E^a_{(C_p,C_d)}(t) = i - t$, when $t \in [1.5i, 1.5(i+1))$.

Definition 8 *Clock drift*: For two clocks C_1 and C_2, the drift of C_2 w.r.t. C_1 at true physical time t is defined by the following function:

$$E^r_{(C_1,C_2)}(t) = \lim_{\delta \to 0^-} \frac{C_2(t+\delta) - C_2(t)}{C_1(t+\delta) - C_1(t)}.$$

Given two analog clocks C_1 and C_2 with change rate functions $f_1(x)$ and $f_2(x)$ respectively, their clock drift can be expressed equivalently by $E^r_{(C_1,C_2)}(t) = \frac{f_2(t)}{f_1(t)}$. For example, $E^r_{(C_p,C_a)}(t) = \frac{t^{\frac{1}{2}}}{1} = t^{\frac{1}{2}}$. It is easy to see that due to the discrete nature of digital clocks, the clock drift between two digital clocks or one digital clock and one analog clock is often unbounded.

In literature, the clock drift is also defined on the basis of the total elapsed times of two clocks (Gupta et al. 1997). In this way, the clock drift between two clocks can always be bounded by clock synchronization in practice. However, this clock drift cannot result in a meaningful property-preservation relation. This is mainly due to the fact that the property-relation between two timed state sequences is determined by local time drifts, which cannot always be bounded by the clock synchronization. A detailed discussion about the choice of the definitions for clock drift can also be found in (Huang 2005).

Next, we introduce the concept of time domain based on the clock concept. Basically, a time domain can be considered as a function assigning time values to untimed behavior. In practice, the action is assigned a time value which is determined by a clock. The clock measures quantitatively the evolution of the time domain. Consequently, the absolute/relative timing differences between behaviors observed in two different time domains can be measured by the deviation and drift between clocks of the two time domains. The following proposition shows this for the relative case. A similar proposition can be proven for the absolute case.

Proposition 1 *Let $\bar{\delta}$ be a state sequence. Further let $\bar{\tau}$ and $\bar{\tau}'$ be two timed state sequences of $\bar{\delta}$ measured by two analog clocks C and C' respectively. If clock drift $E^r_{(C,C')}$ is always in interval $[a,b]$, then $\mathcal{D}^s_r(\bar{\tau},\bar{\tau}') \subseteq [a,b]$.*

Proof Assume δ_i is the i-th state in $\bar{\delta}$, and t_l (t'_l) and t_r (t'_r) are the left-end and the right-end of the time duration of state δ_i counted by clock C (clock C'). Since for all t, $E^r_{(C,C')}(t) \in [a,b]$ (that is, $a \cdot f_C(t) \le f_{C'}(t) \le b \cdot f_C(t)$), we know that $a \cdot (t_r - t_l) \le (t'_r - t'_l) \le b \cdot (t_r - t_l)$. By the definition of function D^s_r, we can see that $\mathcal{D}^s_r(\bar{\tau},\bar{\tau}') \subseteq [a,b]$. □

The following propositions show that it is possible to establish property relations between timing behaviors observed in two time domains with infinitely large clock deviations and drifts. Proposition 2 reveals that an analog clock and a digital clock can always be bridged by a bounded clock drift and deviation if the ratio between corresponding changes of the digital and analog clocks during each clock step is bounded.

Proposition 2 *Let C be a digital clock with step-width Δ and step function $p(i)$, and let C' be an analog clock with change rate $f(t)$. Assume for any i,*

$$\frac{\int_{s(i)}^{s(i+1)} f(x)\,dx}{\Delta} \in [a,b], \quad \textit{where } s(i) = \sum_{j=1}^{i} p(j).$$

Then there exists an analog clock C_a such that $E^a_{(C,C_a)}(t) \in [0,\Delta]$ and $E^r_{(C_a,C')}(t) \in [a,b]$.

Proof Let $r_i = \frac{\int_{s(i)}^{s(i+1)} f(x)\,dx}{\Delta}$. We can construct an analog clock $C_a(t)$ with change rate

$$f_a(t) = \begin{cases} \frac{f(t)}{r_i} & r_i \ne 0 \text{ and } t \in [s(i), s(i+1)), \\ \frac{\Delta}{s(i+1)-s(i)} & r_i = 0 \text{ and } t \in [s(i), s(i+1)). \end{cases}$$

It is easy to check that $\int_{s(i)}^{s(i+1)} f_a(t)\,dt = \Delta$. Therefore, when $t \in [s(i), s(i+1))$, we have

$$\begin{aligned} E^a_{(C,C_a)}(t) &= \sum_{j=0}^{i-1} \int_{s(j)}^{s(j+1)} f_a(x)\,dx + \int_{s(i)}^{t} f_a(x)\,dx - i \cdot \Delta \\ &= \sum_{j=0}^{i-1} \Delta + \int_{s(i)}^{t} f_a(x)\,dx - i \cdot \Delta \le \int_{s(i)}^{s(i+1)} f_a(x)\,dx = \Delta. \end{aligned}$$

At the same time, we have that $E^r_{(C_a,C')}(t) = r_i$ (In the case that $r_i = 0$, we know that $f(t) = 0$ and $f_a(t) = \frac{\Delta}{s(i+1)-s(i)}$. Therefore, $E^r_{(C_a,C')}(t) = 0$.). Now it is easy to see that $E^a_{(C,C_a)}(t) \in [0,\Delta]$ and $E^r_{(C_a,C')}(t) \in [a,b]$. □

Proposition 2 can be used to bridge a digital and an analog clock (which always have an infinitely large drift). In particular the proposition covers the case when the clocks have an infinitely large deviation as well. This is demonstrated by the following example.

Example 4 Let $C(t) = i$ when $t \in [1.5i, 1.5(i+1)]$, and let $C'(t) = t$. It is easy to see that both $E^a_{(C,C')}$ and $E^r_{(C,C')}$ are unbounded. We can construct an analog clock $C_a(t) = 1.5t$. It is easy to see that $E^a_{(C,C_a)} \in [0, 1.5]$ and $E^r_{(C_a,C')} \in [1.5, 1.5]$.

The following proposition covers the case of two analog clocks. Proposition 3 states that two analog clocks can always be bridged by a bounded clock drift and deviation if the ratio between corresponding changes of the two clocks during some fixed period is bounded.

Proposition 3 *Let C be an analog clock with change rate $f(t)$, and let C' be an analog clock with change rate $g(t)$. Further let δ be a positive real. If for any $i \geq 0$ such that*

$$\frac{\int_{i\cdot\delta}^{(i+1)\cdot\delta} g(x)\,dx}{\int_{i\cdot\delta}^{(i+1)\cdot\delta} f(x)\,dx} \in [a, b],$$

then there exists an analog clock C_a such that $E^a_{(C,C_a)} \in [0, \delta]$ and $E^r_{(C_a,C')} \in [a, b]$.

In the following section, we establish the concrete time domains involved in real-time system synthesis. We apply the results of this section to bridge these time domains, such that in the end real-time properties of the realization can be quantitatively predicted from those of the model.

4 Concrete time domains in real-time software synthesis

As we have mentioned previously, the real-time properties of a system are often analyzed in the virtual time domain. In reality, different parts of a real-time system may be timed by different clocks. For example, a real-time controller is usually timed by a digital hardware clock, while the physical environment is usually timed by an analog physical clock. In general, a specific reference clock is chosen, such that real-time properties of different parts can be checked in a uniform time domain of interest (denoted as the reference time domain in this article). Note that the reference time domain can be the true physical time domain mentioned in Sect. 3 or any other physical time domain of interest.

To reason about the real-time properties of a realization based on those of the model, we must establish a proper relation between the timing behavior observed in the virtual time domain and the timing behavior observed in the reference time domain. In this section, we achieve this by constructing several intermediate time domains. To simplify the following discussion, we assume that the realization of the controller is timed by a digital hardware clock and that the reference time is analog. The result based on these assumptions can be easily adapted (based on Propositions 2 and 3) to other situations, e.g. when the reference clock is digital.

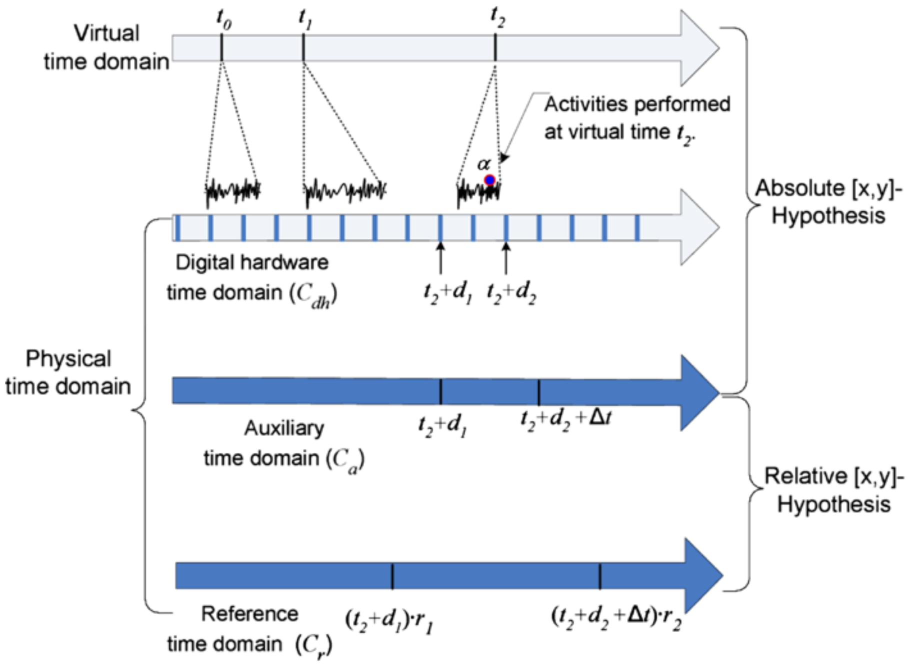
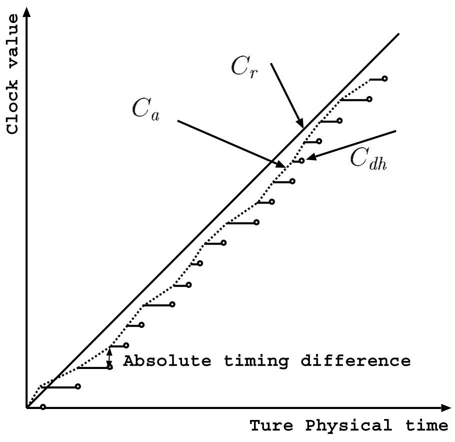

Virtual time domain
t_0
t_1
t_2
Activities performed at virtual time t_2.
α
Absolute [x,y]-Hypothesis
Digital hardware time domain (C_{dh})
t_2+d_1
t_2+d_2
Physical time domain
Auxiliary time domain (C_a)
t_2+d_1
$t_2+d_2+\Delta t$
Relative [x,y]-Hypothesis
Reference time domain (C_r)
$(t_2+d_1)\cdot r_1$
$(t_2+d_2+\Delta t)\cdot r_2$

Fig. 8 The timing relation between different time domains

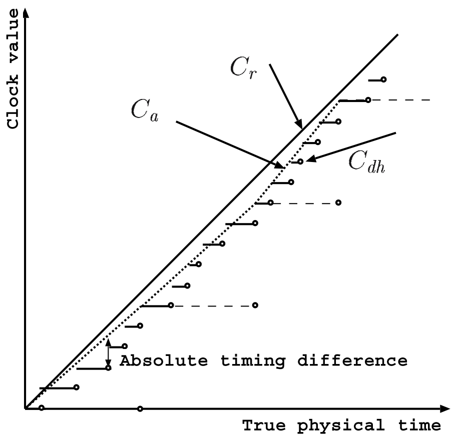
We explore the timing differences between the virtual time domain and the reference time domain by taking several steps (see Fig. 8). We first consider the *absolute* timing differences between the virtual time domain and the digital hardware time domain. Then, we address the timing differences between the digital hardware time domain and the reference time domain. This step is further decomposed into the *absolute* timing differences between the digital hardware time domain and an auxiliary time domain, and the *relative* timing differences between the auxiliary time domain and the reference time domain.

4.1 Timing differences between the virtual time domain and the digital hardware time domain

A model is executed in the virtual time domain, where actions are instantaneous. On the other hand, the realization is executed on a computation platform in the digital hardware time domain, where any action takes up a non-zero number of clock cycles. Due to the uncontrollable physical time, the activation times of actions cannot always be perfectly accurate w.r.t. those specified in the model. Furthermore, due to the techniques that are used to boost the average computing performance of the platform, such as caching, pipelining and memory management techniques, the exact execution times of actions in this digital hardware time domain is difficult to predict. For any action, an *absolute* timing difference (also called *timing deviation*) exists between its virtual time and its digital hardware time. For example, in Fig. 8, if action α is observed at virtual time t_2, we can only guarantee that α is observed within some interval $[t_2 + d_1, t_2 + d_2]$ $(d_1 \leq d_2)$ in the digital hardware time domain.



(a) Hardware step-width



(b) Derived step-widths

Fig. 9 The auxiliary analog clock

4.2 Timing differences between the digital hardware time domain and the reference time domain

The accuracy of a digital hardware clock is affected by discretization and a number of physical factors such as temperature, supply voltage, shock and vibration. As a result, the timing of actions in a realization based on a digital hardware clock is typically not identical to the timing based on a reference clock (e.g. the environmental time or the true physical time). Since the absolute timing differences between them can grow without bounds during execution, it is not effective to measure these timing differences based on the absolute case. On the other hand, the ratio between the change rate of a digital clock and that of an analog clock is unbounded too (see the clock drift discussion in Sect. 3). Therefore it is not effective to measure these timing differences based on the relative case either. To solve this problem, we can construct an auxiliary clock C_a, which has a bounded *deviation* w.r.t. the digital hardware clock C_{dh} and a bounded *drift* w.r.t. the reference clock C_r. A straightforward solution is to construct the auxiliary clock in the same way as the clock (C_a) in the proof of Proposition 2. In this case, $E^a_{(C_{\mathrm{dh}},C_a)}(t) \in [0, \Delta]$, where Δ is the step-width of the digital clock, and $E^r_{(C_a,C_r)}$ is determined by drift factor $r_i = \frac{\int_{s(i)}^{s(i+1)} f(x)\,dx}{\Delta}$, where $f(x)$ is the change rate function of C_r. Figure 9(a) shows an example of the auxiliary clock C_a for a digital hardware clock C_{dh} and a reference clock C_r. In Fig. 9(a), the reference clock is assumed to be the true physical time. In this case, the relative timing differences between C_r and C_a are the slopes of the dotted segments in the figure.

Example 5 Reconsider Example 1 and suppose that the third action of the controller (sending out the second letter 'E') in the digital hardware time domain is observed in time interval $[0.12, 0.12 + 0.001]$. Assume further that the speed of the digital hardware clock C_{dh} is 100 MHz (the step-width of C_{dh} is 10^{-8} seconds). Then we can predict that the action is observed in time interval $[0.12, (0.12 + 0.001) + 10^{-8}]$

in the auxiliary time domain. Suppose that the drift factor r_i of C_{dh} can be up to 50% (faster or slower) w.r.t. reference clock C_r. In other words, during each step of clock C_{dh}, the average change rate of clock C_r is at most 1.5 and at least 0.5 times faster than that of clock C_{dh}. Now, we can predict that the action is observed in time interval $[0.12 \times 0.5, (0.12 + 0.001 + 10^{-8}) \times 1.5]$ in the reference time domain.

However due to physical uncertainties, the clock drift factors $r_i = \frac{\int_{s(i)}^{s(i+1)} f(x)\,dx}{\Delta}$ can be very large. As a result, the properties predicted for the realization can be unacceptably far from those of the model. In the following we propose a solution to obtain better prediction results.

The auxiliary clock mentioned previously is based directly on the step-width of the digital hardware clock. But the auxiliary clock can also be constructed by choosing a larger step-width (as shown in Fig. 9(b) where the derived step-width used to construct the auxiliary clock consists of 5 digital hardware clock steps). This results in a smaller relative time difference. To see this, assume we have digital hardware clock C with step-width Δ and function $s(i)$. Let $r_i = \frac{\int_{s(j)}^{s(j+1)} f(x)\,dx}{\Delta}$ and assume that the step-width used to construct the auxiliary clock consists of k steps of C. Then we have that $r_i' = \frac{\int_{s(k*i)}^{s(k*(i+1))} f(x)\,dx}{k*\Delta} = \frac{1}{k} \cdot (r_{k\cdot i} + r_{k\cdot i+1} + \cdots + r_{(k+1)\cdot i-1})$. It is easy to prove that $\min\{r_j \mid k \cdot i \le j \le (k+1) \cdot i - 1\} \le r_i' \le \max\{r_j \mid k \cdot i \le j \le (k+1) \cdot i - 1\}$. Consequently, the range of r_i' is always within the range of r_i.

As an example reconsider the IEEE flash controller. Assume that the step-width used to construct the auxiliary clock consists of 10^8 digital hardware steps (1 second in the digital hardware time domain). For most digital hardware clocks used in PC platforms, the relative clock drift is around 10^{-5} seconds per second. So we can safely assume the timing drift to be bounded by 10^{-3}. At the same time, we assume that the absolute timing deviations between the digital hardware and the auxiliary clock should be no more than 10^{-3} seconds. Therefore, we can predict that the action of sending out the second letter 'E' is observed in time interval $[(0.12 - 10^{-3}) \times (1 - 10^{-3}), (0.12 + 0.001 + 10^{-3}) \times (1 + 10^{-3})]$ in the reference time domain. This yields a much tighter prediction result than in the previous case.

5 Hypotheses for real-time software synthesis

In the previous section, we have established the relation between the observation times of an action in the virtual time domain and in the reference time domain. This is accomplished by building a series of (absolute and relative) intermediate timing domains (see Fig. 8). In this section, we summarize these timing differences into two parameterized hypotheses. More specifically, the absolute $[x, y]$-hypothesis captures two absolute timing differences: one between the virtual and digital hardware time domains and the other between the digital hardware and auxiliary time domains. The relative $[x, y]$-hypothesis captures the timing differences between the auxiliary and reference time domains. Consequently, if both hypotheses are satisfied, real-time properties of the realization can be predicted from those of the model in two steps.

5.1 The absolute $[x, y]$-hypothesis

We showed that an action, which is observed at a certain time point in the virtual time domain is typically observed at another time point in the auxiliary time domain. Without careful treatment, the deviation between the observation times of an action in the virtual time domain and in the auxiliary time domain can accumulate without bounds during system execution. Even worse, these timing deviations may lead to a different execution order of actions between the model and the realization, which may result in faulty behaviors (see Fig. 1(b)). To avoid these problems, we propose the *absolute* $[x, y]$*-hypothesis* to bound the absolute timing difference between the model in the virtual time domain and the realization in the auxiliary time domain. By doing so, we can predict the real-time properties of the realization (in the auxiliary time domain) from those of the model. This is elaborated in the following.

We can use a timed action sequence to represent an execution of a real-time system (either a model or a realization). Each timed action sequence can be considered as a sequence of timed actions (α_i, t_i), where t_i is the observation time of action α_i. A timed action sequence can alternatively be represented as a pair of sequences, an action sequence $(\alpha_0\alpha_1\alpha_2\ldots)$ and a time sequence $(t_0t_1t_2\ldots)$.

The absolute $[x, y]$-hypothesis requires that for any timed action sequence $\bar{\tau}_\alpha$ of the realization in the auxiliary time domain, the following conditions are satisfied:

(1) *There exists a timed action sequence* $\bar{\tau}'_\alpha$ *of the model in the virtual time domain, such that* $\bar{\tau}_\alpha$ *and* $\bar{\tau}'_\alpha$ *share the same action sequence* $\bar{\alpha}$.
(2) *For each action in* $\bar{\alpha}$*, the deviation between its observation times in* $\bar{\tau}_\alpha$ *and in* $\bar{\tau}'_\alpha$ *must be within interval* $[x, y]$.

Parameters x and y in the hypothesis represent a lower respectively upper bound of the absolute timing difference between two time domains.

In Sect. 2.3, we showed that a real-time property relation between two timed *state* sequences can be established based on their absolute timing differences (see Theorem 1). This relation reveals that real-time properties can be preserved (up to a deviation $y - x$) between two absolute $[x, y]$-close timed *state* sequences (or real-time systems). The absolute $[x, y]$-hypothesis assumes that timed *action* sequences in the auxiliary time domain are close to those in the virtual time domain based on their absolute timing differences. Therefore, we need to examine property relations between timed *action* sequences. This is achieved by the following line of reasoning.

- *Convert timed action sequences to timed state sequences:* A timed execution trace of a system can be represented by either a timed state sequence or a timed action sequence. It is easy to encode one form of representation into the other. One possibility to encode a timed action sequence into a timed state sequence is by letting the observation of each action be an instantaneous state, and inserting a new duration state Φ between any two instantaneous states (Nicola and Vaandrager 1990). For example, suppose a timed action sequence $\bar{\tau}_\alpha$ is

$$(\alpha_1, t_1)(\alpha_2, t_2)(\alpha_3, t_3)\ldots(\alpha_i, t_i)\ldots.$$

A corresponding timed state sequence $\bar{\tau}_\delta$ is

$$(S_{\alpha_1}, [t_1, t_1])(\Phi, I_1)(S_{\alpha_2}, [t_2, t_2],)\ldots(S_{\alpha_i}, [t_i, t_i])(\Phi, I_i)\ldots,$$

where I_i is $[t_i, t_i]$ if $t_i = t_{i+1}$ or $[t_i, t_{i+1})$ if $t_i < t_{i+1}$. Here S_{α_i} is a state containing the single atomic proposition "α_i is observed" and Φ is a state at which no atomic proposition is observed.

- *Relate the proximity between timed action sequences and the proximity between their corresponding timed state sequences:* Let $\bar{\tau}_\alpha$ and $\bar{\tau}'_\alpha$ be two timed action sequences, which share the same action sequence $\bar{\alpha}$. Assume $\bar{\tau}_\delta$ and $\bar{\tau}'_\delta$ to be their corresponding timed state sequences. If $\bar{\tau}'_\alpha$ is absolute $[x, y]$-close to $\bar{\tau}_\alpha$, then we can easily derive that $\bar{\tau}'_\delta$ is absolute $[x, y]$-close to $\bar{\tau}_\delta$ (see proximity function D_a^s in Sect. 2.1). Consequently, by Definition 2, we know that if the absolute $[x, y]$-hypothesis is satisfied, then the realization in the auxiliary time domain is absolute $[x, y]$-close to its model in the virtual time domain.
- *Preserve correctness between timing behaviors:* By Theorem 1, if the realization is absolute $[x, y]$-close to its model, and the model satisfies *MTL* property P, then the realization satisfies property $R_a^{(y-x)\Box}(P)$. Property $R_a^{(y-x)\Box}(P)$ has the same form as P, but its quantitative timing bounds have an absolute deviation of $y - x$ from those of P (see Sect. 2.3). Hence, real-time properties that hold in the virtual time domain can be preserved in the auxiliary time domain by up to a deviation of $y - x$, if the absolute $[x, y]$-hypothesis is satisfied.

In Sect. 6.3, we show how the absolute $[x, y]$-hypothesis can be incorporated in a concrete synthesis tool.

Example 6 Reconsider Example 1. Suppose that the absolute $[x, y]$-hypothesis is satisfied when synthesizing the *IEEE* flash board controller software and that $[x, y] = [0, 0.001]$. This indicates that the observations of actions in the auxiliary time domain are always later than their corresponding observations in the virtual time domain, but that the delay never exceeds 0.001 seconds. At the same time, the output letters on the flash board are always displayed in the correct order in the realization. The error observed in Fig. 1(b) will not occur in this case. Furthermore, it is easy to check that the model satisfies quantitative real-time property φ_v stating that the 'IEEE' word always appears 2.7 seconds after it has been erased. φ_v can be formalized by *MTL* expression $\Box(q \to \Diamond_{[2.7,2.7]} p)$, where p represents that the 'IEEE' word appears and q represents that the 'IEEE' word is erased.

Now we can predict that the realization satisfies the following property φ_a in the auxiliary time domain, which deviates 0.001 from property φ_v (see Sect. 2.3).

$$\varphi_a = R_a^{0.001\Box}(\varphi_v) = \Box(p \to \Diamond_{[2.699,2.701]} q).$$

Formula φ_a states that the realization of the controller in the auxiliary time domain always displays the 'IEEE' word between 2.699 and 2.701 seconds after the word is erased.

5.2 Relative $[x, y]$-hypothesis

In practice, clock drift often exists between the auxiliary clock and the reference clock. To predict the real-time properties of the realization in the reference time domain based on those in the auxiliary time domain, we propose the relative $[x, y]$-hypothesis, which requires that:

The ratio R between the change rates of auxiliary clock C_a and reference clock C_r must be within interval $[x, y]$, where x (y) is a lower (upper) bound of R.

If the relative $[x, y]$-hypothesis is satisfied by the target platform, it is easily seen that the ratio between the change rates of reference clock C_r and auxiliary clock C_a should be within interval $[\frac{1}{y}, \frac{1}{x}]$.

The clock drift of C_r w.r.t. C_a can be used to estimate the relative timing difference between timing behaviors interpreted in the reference time domain and in the auxiliary time domain, which is illustrated in Example 7. If the relative $[x, y]$-hypothesis is satisfied, the realization in the reference time domain is relative $[\frac{1}{y}, \frac{1}{x}]$-close to the realization in the auxiliary time domain. Consequently, we can predict real-time properties of the realization in the reference time domain from those in the auxiliary time domain based on Theorem 2.

Example 7 Reconsider the *IEEE* flash board controller in Example 1. Assume the change rate of the auxiliary clock can deviate w.r.t. that of the reference clock by up to 0.5% (faster or slower). In this case, we can calculate that the ratio of the change rates between the reference clock and the auxiliary clock is within interval $[\frac{1}{1.005}, \frac{1}{0.995}]$. Consequently, we can predict that the realization of the *IEEE* flash board controller satisfies real-time property φ_{re} in the reference time domain, where

$$\varphi_{re} = R_r^{[1/1.005, 1/0.995]}(\varphi_a) = \Box(q \to \Diamond_{[2.699/1.005, 2.701/0.995]} p).$$

Since $[2.699/1.005, 2.701/0.995] \subset [2.685, 2.715]$, φ_{re} is stronger than property $\Box(q \to \Diamond_{[2.685, 2.715]} p)$. This indicates that the realization of the controller in the reference time domain always sends out the 'IEEE' word t seconds ($t \in [2.685, 2.715]$) after the word is erased.

From the example above, we can see that the values of x and y in the absolute $[x, y]$-hypothesis and in the relative $[x, y]$-hypothesis have a direct impact on the preservation of quantitative real-time properties. This is further illustrated by the following example, in which both hypotheses are involved.

Example 8 Assume some model M to satisfy a deadline property P_M: $\Box(p \to \Diamond_{[2,3]} q)$ indicating that stimulus p is always followed by response q within 2 to 3 seconds. Realization R is synthesized from M respecting both the absolute $[x_a, y_a]$- and the relative $[x_r, y_r]$-hypothesis. Then we know that R satisfies property P_R:

$$P_R = R_r^{[1/y_r, 1/x_r]}(R_a^{(y_a - x_a)\Box}(P_M)) = \Box(p \to \Diamond_{[(2-y_a+x_a)/y_r, (3+y_a-x_a)/x_r]} q).^3$$

The example above indicates that the quantitative difference between real-time properties of a model and its realization can be reduced by changing the values of either $y_a - x_a$, x_r or y_r. The value of $y_a - x_a$ is affected by the model itself, the scheduling algorithm and the computational capabilities of the target platform (see

[3] If $2 - (y_a - x_a) < 0$, then $2 - (y_a - x_a)$ is replaced by 0. More information can be found in (Huang 2005).

Sect. 6.3), while the values of x_r and y_r can be estimated by the drift factor of the auxiliary clock w.r.t. the reference clock.

The above reasoning can also be applied in a reverse way. For example, the drift of the auxiliary clock w.r.t. the reference clock is independent from the design process and can be pre-measured. Assume x_r and y_r are 0.99 and 1.02 respectively, and assume we require property P_R ($\Box(p \to \Diamond_{[3,6]}q)$) to be satisfied by the realization. In this case, the model should satisfy property $P_M = \Box(p \to \Diamond_{[1.02\times 3+y_a-x_a, 0.99\times 6-(y_a-x_a)]}q)$. The smaller the value of $y_a - x_a$ is, the weaker property P_M will be. By estimating the value of $y_a - x_a$ (see also Sect. 7.3), one can know beforehand that the model should satisfy property P_M, in order to correctly deploy the system on the target platform.

6 Correctness-preserving software synthesis

In previous sections, we explained ineliminable timing differences between a model and its corresponding realization. Furthermore, we proposed two hypotheses to capture timing inconsistencies and to eliminate functional inconsistencies between a model and its realization. In this section, we show that both hypotheses (but especially the absolute $[x, y]$-hypothesis) can be incorporated into a concrete synthesis tool.

In the following, we first give a brief overview of the modeling language POOSL (Parallel Object-Oriented Specification Language) (Geilen et al. 2001). Our discussion about POOSL focuses on its execution mechanism, which generates timed action sequences for POOSL models. Following that, we show that the absolute $[x, y]$-hypothesis is supported by the synthesis tool Rotalumis, which converts POOSL models into realizations.

6.1 POOSL

The POOSL language integrates a process part based on a timed and probabilistic extension of CCS and a data part based on the concepts of traditional object-oriented languages (Geilen et al. 2001). A POOSL model consists of a set of parallel processes, which perform their activities asynchronously and communicate with each other synchronously by message passing. Each process can call and execute its methods which are formed by the statements in Table 1. Different from procedure or functions used in imperative programming languages, the process methods in POOSL allow tail-recursion to specify infinite behaviors in a succinct way.

The formal semantics of POOSL can be found in (van der Putten and Voeten 1997; Geilen 2002; van Bokhoven 2002). The formal semantics of POOSL given in (van der Putten and Voeten 1997) addresses untimed behaviors covering parallelism, communication, non-determinism and data. The semantics of the timed language (where data is abstracted from) is given in (Geilen 2002). This work also introduces an execution mechanism for the language and proves it to be correct with respect to the semantics of the language. The most complete formal semantics (including time, probabilities and data) of the POOSL language can be found in (van Bokhoven 2002). The formal semantics of the POOSL language is beyond the scope

Table 1 POOSL statements

$S ::=$	E	expression	$[E_c]S$	guarded execution
	$m(E_1, \ldots, E_i)(v_1, \ldots, v_j)$	methods call	**interrupt** S_1 **with** S_2	interrupt
	par S_1 **and ... and** S_n **rap**	parallel composition	**abort** S_1 **with** S_2	abort
	S_1**;** S_2	sequential composition	**delay** E	time synchronization
	$ch!m(E_1, \ldots, E_i)\{E\}$	message send	**if** E_c **then** S_1 **else** S_2 **fi**	choice
	$ch?m(v_1, \ldots, v_i \| E_c)\{E\}$	(conditional) message receive	**while** E_c **do** S **od**	loop
	sel S_1 **or ... or** S_n **les**	non-deterministic selection	**skip**	empty behavior

of this article, and readers are referred to the above references for more details. The POOSL language has been successfully applied to the modeling and the analysis of many industrial systems, such as network processors (Theelen et al. 2003; Noonan and Flanagan 2004), a multimedia application (van Wijk et al. 2003) and an Internet router (Theelen et al. 2001).

In the following subsection, we focus on the execution mechanism for POOSL models. This execution mechanism is also incorporated in Rotalumis to ensure that the semantics of the realization respects the semantics of the model.

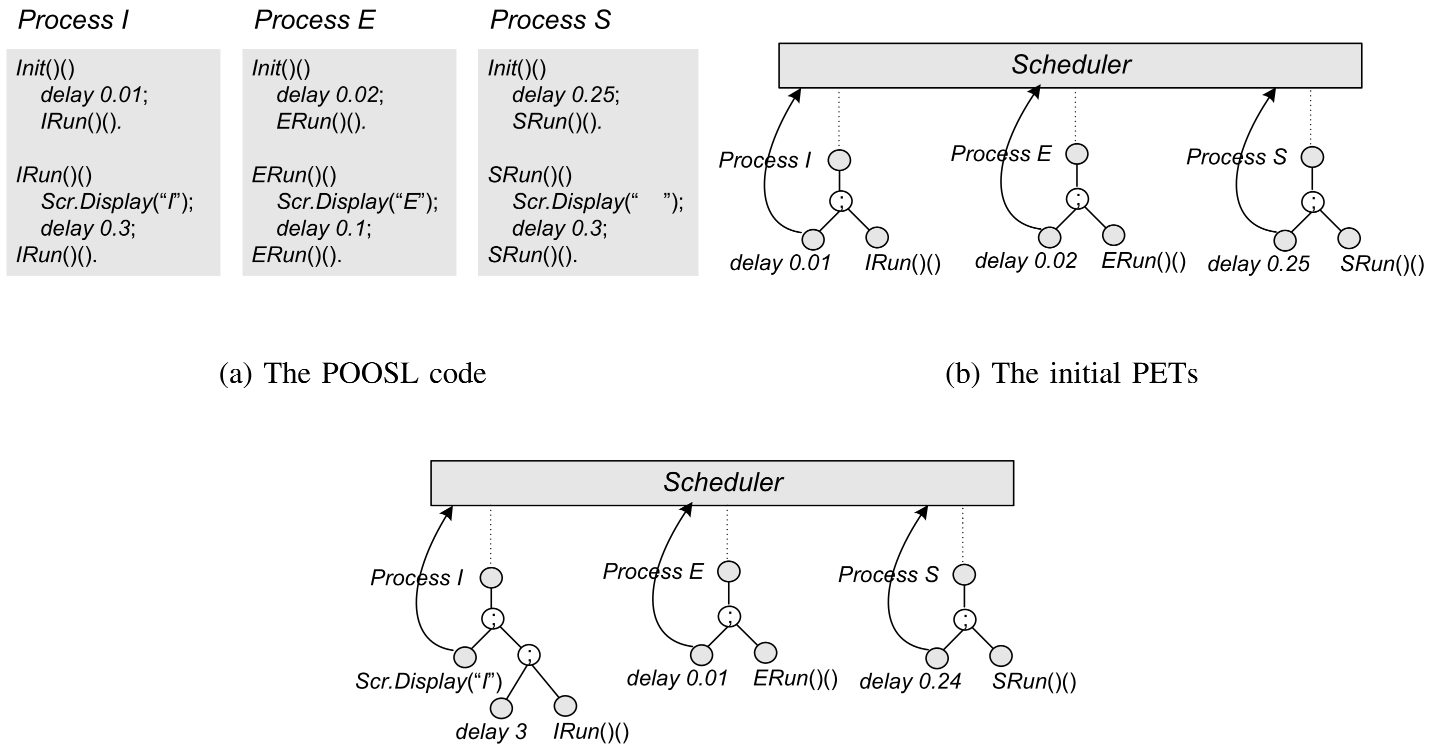
6.2 The PET scheduler

In the execution mechanism of POOSL models, each process is represented by a process execution tree (PET), and the model is executed by a PET scheduler which chooses available actions from these trees to be executed. For example, Fig. 10(a) shows the POOSL model of an IEEE flashboard controller consisting of three parallel processes I, E and S. The PET of each process is given in Fig. 10(b). Each leaf of a PET is a statement (such as *delay 0.01*) or a (recursively defined) process method (such as *IRun*()()). During the evolution of the system, each PET provides its statements available for execution (such as *delay* 0.25 of Process S in Fig. 10(b)) to the PET scheduler and dynamically modifies its tree according to the choice made by the PET scheduler. For example, after having performed the *delay 0.01* statement, the PETs modify their trees from Fig. 10(b) to Fig. 10(c), in which data method *Scr.Display("I")* can output a letter 'I' to the screen.

The PET scheduler plays a central role during execution. Next we zoom into the PET scheduler to explain how the concurrent real-time behavior is executed.

As we have mentioned before, concurrent processes in a POOSL model are represented by a set of PETs, in which each leaf node represents a statement. These PETs offer their statements to be executed to the PET scheduler, by inserting the corresponding nodes into a list of the PET scheduler.[4] The PET scheduler maintains two different lists, a delay list (for all delay statements) and an action list (for all other statements). The scheduler grants the execution of statements by giving action statement higher priority than delay statements. Since actions are instantaneous in the

[4]Message send and receive statements are an exception; they are not inserted until their counterparts are ready too.



```
Process I

Init()()
    delay 0.01;
    IRun()().

IRun()()
    Scr.Display("I");
    delay 0.3;
IRun()().
```

```
Process E

Init()()
    delay 0.02;
    ERun()().

ERun()()
    Scr.Display("E");
    delay 0.1;
ERun()().
```

```
Process S

Init()()
    delay 0.25;
    SRun()().

SRun()()
    Scr.Display("    ");
    delay 0.3;
SRun()().
```

(a) The POOSL code

(b) The initial PETs

(c) The PETs after the first *delay 0.01*

Fig. 10 The PETs of the IEEE flashboard controller

```
1:  StatementList actions, delays; Real $T_v$; // global variables
2:  PET_scheduler() {
3:      Statement *action, *delay;
4:      $T_v = 0$; // Initialise the virtual time
5:      PETs_startup(); // PETs insert their initial statements to the action and delay lists
6:      while (true) do {
7:          while (Get_from_action_list(action)) do // Choose a statement from the list
8:              if (actions→Grant()) then{ // The grant is accepted, the statement is executed and new statements are inserted into the lists
9:                  Remove_from_action_list(action); // Remove the current statement and reset all statements in the list available
10:             }
11:             else // statement refuses the grant permission
12:                 Set_action_unavailable(action); // The current statement is not available for the following action selections
13:         }
14:         if (Get_from_delay_list(delay)) then {
15:             delay→Grant(); // Reduce the duration of each delay statement in the list by the duration of the granted delay, add new statements to the lists
16:             $T_v := T_v$+delay→duration; // The virtual time progresses
17:             Remove_from_delay_list(delay); // Remove the current statement and reset all statements in the list available
18:         }
19:         else
20:             Deadlock();
21:     }
22: }
```

Fig. 11 The PET scheduler

virtual time domain, the virtual time does not progress until all executable actions are performed.[5]

The behavior of the PET scheduler is shown in Fig. 11. When the scheduler grants a statement (line 8), the corresponding node checks whether it is interrupted or guarded. Only if it is not interrupted and all its guards are open, the node accepts the grant and carries out its statement. In the other case, the node refuses the grant and the scheduler picks another statement in the list. When a node has actually performed its statement, its PET adjusts the tree structure, withdraws statements from the list[6] and provides the new statements to be executed to the scheduler (line 9). Since the execution of a statement may insert new statements into the action list or may change the status of interruptions or guards, the nodes that refused a previous grant may be available again for the next choice of the PET scheduler. The scheduler repeatedly grants the statements in the action list until there is no statement anymore that accepts the grant. Then, the scheduler grants the first delay statement in the delay list, where delay statements are ordered according to their expiration times. When a

[5] Some actions in the list may not be executable because a corresponding guard may be closed or the action may be interrupted.

[6] For instance, assume a tree constrains the following non-deterministic choice **sel** $x := x + 1$ **or** $y := y + 1$ **les**. If $x := x + 1$ is granted by the scheduler, then $y := y + 1$ should be withdrawn from the list.

delay statement is granted, the virtual time progresses by the duration of the delay statement (line 16), and new statements are inserted into the lists. At the same time, the durations of the remaining delay statements in the list are adjusted accordingly. When the scheduler finds no delay statement to grant, the system is terminated or enters a deadlock state (line 20).

In the next subsection, we show how Rotalumis extends the PET scheduler to comply with the absolute $[x, y]$-hypothesis.

6.3 Rotalumis

Rotalumis takes as input a POOSL model built during system modeling and automatically generates the executable realization for the target platform. In this subsection, we show how the Rotalumis tool supports the absolute $[x, y]$-hypothesis. The following techniques are adopted in Rotalumis.

(1) *Process execution trees:* The POOSL language provides ample facilities to describe system characteristics such as parallelism, nondeterministic choice and communication that are not directly supported by implementation languages such as C, C++ and Java. In order to provide a correct and smooth mapping from a POOSL model to a C++ realization, PETs are used to bridge the gap between the semantics of the two languages. The data part of a POOSL model is transformed into byte code interpreted by Rotalumis during run-time. The process part of a POOSL model is transformed into a set of PETs implemented in C++.

The PETs in the realization have the same (virtual) time semantics as their counterparts in the model. The PET scheduler in Rotalumis also behaves the same as the PET scheduler used to execute the POOSL model. It schedules the PET nodes based on their (virtual) time semantics. As a result, the realization exhibits exactly the same behavior as the model, if interpreted in the virtual time domain.

Since the progress of the virtual time is monotonically increasing, which is consistent with the progress of the auxiliary time, the action order observed in the virtual time domain is consistent with that in the auxiliary time domain. Therefore, the PET scheduler in Rotalumis ensures that for any timed execution of the realization, a corresponding timed execution in the POOSL model can always be found, such that both executions exhibit the same observable action sequence. Therefore, the first requirement (see Sect. 5.1) of the absolute $[x, y]$-hypothesis is guaranteed by the synthesis tool.

(2) *Synchronization between the virtual time and the digital hardware time:* To minimize the timing inconsistencies between the realization interpreted in the digital hardware and virtual time domains, the PET scheduler in Rotalumis tries to schedule these actions in the digital hardware time domain according to their time of occurrence in the virtual time domain. To achieves this, the initial value of the digital hardware time is stored in T_d by accessing the hardware clock. Therefore, $T_d := \mathit{Read\_real\_time}()$ is inserted between lines 4 and 5 in Fig. 11. Notice that the virtual time is still kept as a "reference" in the PET scheduler of Rotalumis. For instance, when a delay statement with duration t is granted by the PET scheduler, the virtual time T_v advances to $T_v + t$. Correspondingly, in

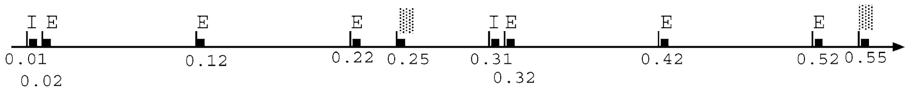


Fig. 12 The Rotalumis realization of IEEE flash controller

the digital hardware time domain, the PET scheduler of Rotalumis should keep on checking whether the digital hardware clock has the offset $T_v + t$ w.r.t. its initial digital hardware time, before it starts to grant statements in the action list. Therefore, the following code is inserted between lines 17 and 18 in Fig. 11.

$$\text{while } (\mathit{Read\_real\_time}() - T_d <= T_v) \text{ do}\{\}$$

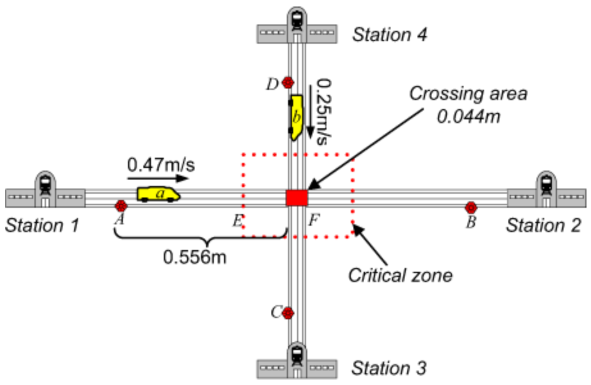
Consequently, we can see that the delay statement with duration t in Rotalumis is interpreted as *no more* than t in the digital hardware time domain to compensate for the execution times of actions and scheduling overhead. This is different from the traditional interpretation in implementation languages such as Java, C and C++, where the interpretation is *at least* t. In this way, accumulated timing errors between the model and its realization are minimized. For instance, when Rotalumis maps the IEEE flashboard controller onto the target platform, the PET scheduler schedules actions from three processes according to their virtual times. Compared to the realization generated by TAU G2 (illustrated in Fig. 2(b)), the realization generated by Rotalumis (illustrated in Fig. 12) not only avoids incorrect outputs but avoids accumulated timing errors as well.

The timing errors between the model and its realization in the digital hardware time domain can be estimated to lie in certain interval $[a, b]$ (see the discussion in Sect. 7.3). The timing deviation of the *analog* hardware clock w.r.t. the *digital* hardware clock is determined by the quality of the digital hardware clock and the step-width chosen for constructing the auxiliary clock. As we have discussed in Sect. 4.2, we can assume that the time errors are within interval $[-\delta, \delta]$. For commonly-used oscillator clocks in PCs, δ is no larger than 10^{-3}. For higher quality oscillators, the time errors can be assumed to lie in a much tighter interval. Therefore the execution of the realization in the auxiliary time domain is absolute $[a - \delta, b + \delta]$-close to a trace of the model. Hence the tool complies with the second requirement (see Sect. 5.1) of the absolute $[x, y]$-hypothesis.

Combining items (1) and (2), we conclude that the absolute $[x, y]$-hypothesis is complied with by the synthesis tool. The quality of the digital hardware clock and the step-width chosen for constructing the auxiliary clock (see Sect. 4.2) determines whether the relative $[x, y]$-hypothesis is complied with as well.

It should be noticed that the satisfaction of both hypotheses does not always imply a correct realization. However, the parameters of both hypotheses allow designers to quantitatively predict the real-time properties of the realization. Based on this information, the correctness of the realization can be judged. If the real-time properties of the realization are relaxed too much, designers can choose a faster platform to reduce the execution times of actions and the scheduling overhead, they can use a more stable hardware clock, or refine the model itself.

Fig. 13 A railroad crossing system



7 A case study

In this section, we use a railroad crossing system to demonstrate how a real-time software controller can be generated from its POOSL model while preserving the correctness.

7.1 System description

The railroad crossing system that we choose is similar to the standard railroad crossing problem used to compare different formal frameworks for modeling real-time systems (Heitmeyer et al. 1993). As shown in Fig. 13, four stations are connected by two orthogonal tracks. Train a (b) runs back and forth between stations 1 and 2 (and stations 3 and 4). Four sensors (A, B, C and D) are installed at some distance to the crossing to detect the passing of the trains. To compensate for the deceleration time of the train and to avoid the stopping of the train inside the crossing area, a critical zone is defined as shown in Fig. 13. When a train approaches a border of the critical zone from outside of the zone, it has to request permission to enter the crossing. If the request is denied, the train has to stop and wait until permission is granted. When the train leaves the crossing, it has to release the crossing. As a consequence, the crossing is free and available for the other train to enter. We assume that the speed of the train is constant between two sensor points (e.g. from point A to B) if no stop occurs.

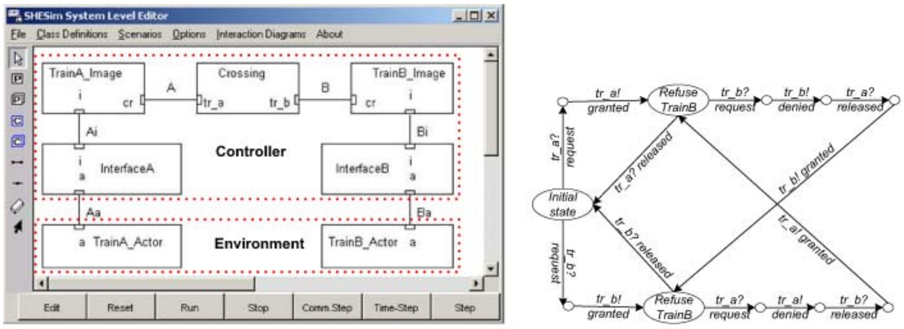
In our case, the physical system is built by LEGO materials, where the relevant parameters are given in Table 2.[7] In this article, we focus on the synthesis of the model into the realization.[8] The preservation of correctness is demonstrated by the preservation of an important quantitative real-time property P, which specifies that the trains should never collide and the system should operate as efficient as possible at the same time. The definition of P will be formally presented in the next subsection.

[7] In the table, the deceleration distance of a train refers to the distance from the point where the train starts to decelerate to the point where the train stops completely.

[8] For a more detailed description of the complete system design including requirement analysis, step-wise model refinement and system synthesis, interested readers are referred to (Huang 2005).

Table 2 Parameters of the railroad crossing system

	Para.	Value	Meaning
Crossing	l_{sc}	0.556 m	The distance from each sensor to the nearest border of the crossing
	l_c	0.044 m	The size of the crossing
Train	l_t	0.198 m	The length of the train
	v_a	0.47 m/s	The speed of train *a*
	v_b	0.25 m/s	The speed of train *b*
	D_a	0.045 m	The deceleration distance of train *a*
	D_b	0.015 m	The deceleration distance of train *b*



(a) A POOSL model

(b) The state-action diagram of the *Crossing* process

Fig. 14 The model of the railroad crossing system

7.2 System modeling

In this section, we introduce a parallel solution to design the railroad crossing system, which is modeled by the POOSL language. Figure 14(a) shows the POOSL model of the system, where the system consists of two parts, a controller and an environment. The controller consists of five parallel processes: a *Crossing*, two train image processes (*TrainA_ Image* and *TrainB_Image*) and two interface processes (*InterfaceA* and *InterfaceB*). The environment consists of two train actor processes: *TrainA_Actor* and *TrainB_Actor*, which model the behavior of the physical trains. The interactions between these processes are accomplished through ports connected by static channels. For example, port *tr_a* of the *Crossing* process and port *cr* of the *TrainA_Image* process are connected by channel A. In the following, we give a more detailed explanation of each process in the controller.

(a) *The Crossing:* The *Crossing* process is used to control the access to the physical crossing based on the requests from the train images. It is required that:

- No collision should occur between two trains. The *Crossing* should grant access to the physical crossing to at most one train at a time.

```
Check_leftsensor()(); //polling the sensor state till receiving a sensor signal
delay ($T_1$); // the duration from the train passing the 1st sensor to reaching the border of the critical zone
cr!request;
sel
  cr?granted;
  i!continue;
  delay ($T_2$); // the duration from the train entering the critical zone to leaving the crossing without stopping
or
  cr?denied;
  i!pause;
  cr?granted;
  i!resume;
  delay ($T_3$); // the duration from the train restarting in the critical zone to leaving the crossing area
les;
cr!release;
```

Fig. 15 Behavior of the train image moving from left to right

- The system should be deadlock free. The *Crossing* should never block access to both trains at the same time.

In addition, to ensure the operation efficiency, the *Crossing* should avoid unnecessary waiting of the trains. Therefore, in our solution, the *Crossing* process always grants the request to access the physical crossing immediately when it is aware that the physical crossing is free. Figure 14(b) shows a state-action diagram of the *Crossing*, where *tr_a?request* indicates that the *Crossing* receives a *request* message through port *tr_a* and *tr_a!granted* indicates that the *Crossing* sends out a *granted* message through port *tr_a*. When the *Crossing* is at the *Initial state*, the physical crossing is assumed to be free. If the *Crossing* receives an access request from a train image (e.g. *tr_a?request*), it immediately grants the request (e.g. *tr_a!granted*) and enters into a state (*Refuse TrainB* or *Refuse TrainA*) where the request from the other train is denied. At this state, if the physical crossing is released (e.g. *tr_a?released*) before the request from the other train arrives, the *Crossing* returns to the *Initial state*. Otherwise, the request from the other train is denied immediately and will be granted when the physical crossing is released.

(b) *The train image processes:* Each *train image* (e.g. *TrainA_Image* or *TrainB_Image*) refers to a physical train and can be considered as its mirrored image inside the controller. A train image monitors the states of a physical train and controls the physical train according to its states. For example, Fig. 15 shows a piece of code, which describes a part of the behavior of the train image, starting from the moment when it receives a message from the first sensor until the moment when it releases the physical crossing. T_1, T_2 and T_3 are used to specify the timing relations between actions, which are estimated according to relevant information (such as the speed of the train and the size of the critical zone). In our case, T_1 is 1.015 seconds and T_2 is 0.7 seconds for *TrainA_Image*. If the request from a train image to access the crossing is granted, the train image occupies the crossing for T_2 seconds before it releases the crossing. If the request to access the crossing is denied, the train image stops the physical train immediately and waits for the *granted* message to restart the physical train.

(c) *The interface processes:* The interface processes establish the communication between the control processes (train images) and the physical environment (train actors).

(d) *Real-time property P:* Now we can specify property P w.r.t. the design solution we have adopted. We assume that the speed of each physical train is constant between two sensors (such as A and B in Fig. 13), if no stop occurs. When the train image receives the first sensor signal on each journey, it can estimate time t at which it should send the request to access the physical crossing. Time t should be chosen appropriately. Sending the request too early can increase the occupancy time of the crossing and decrease the efficiency of the system when the request is granted. On the other hand, sending the request too late can lead to the physical train stooping inside the physical crossing and causing a collision when the request is denied. In our example, t is required to be in interval $[1, 1.025]$ for *TrainA_Image*. Furthermore, suppose that the corresponding physical train of *TrainA_Image* takes at most 1.7 seconds to pass the physical crossing. Then, property P can be formalized using the following *MTL* formula:

$$P = \Box(p_a \rightarrow ((\Diamond_{[1,1.025]} r_a \wedge \Diamond_{[1,1.025]} q_a) \vee \Box_{[0,1.7]} \neg r_a)),$$

where p_a, q_a and r_a are given as follows:

- p_a : *TrainA_Image* receives the first sensor signal on one journey.
- q_a : *TrainA_Image* sends out message *pause*.
- r_a : *TrainA_Image* receives message *denied*.

Formula P states that after *TrainA_Image* receives the first sensor signal, **either** both r_a and q_a are observed within interval $[1, 1.025]$ **or** the physical train passes the crossing without stopping.

Under the assumption that actions are instantaneous, a stronger property P_v is satisfied in the model,[9] which is given as follows:

$$\Box(p_a \rightarrow ((\Diamond_{[1.015,1.015]} r_a \wedge \Diamond_{[1.015,1.015]} q_a) \vee \Box_{[0,1.7]} \neg r_a)).$$

Here, we give an informal explanation that P_v is indeed satisfied by the model. Let us look at the code in Fig. 15. After *TrainA_Image* receives the first sensor signal on one journey (p_a is observed), it waits for 1.015 (T_1) seconds. After 1.015 seconds, *TrainA_Image* sends a *request* message to the *Crossing*. Note that all interactions are instantaneous in the virtual time domain and the crossing replies to *TrainA_Image* without any time delay (see Fig. 14(b)). In the case that the request is denied, *TrainA_Image* gets the denied message and sends out the pause message after exactly 1.015 seconds. In this case, $p_a \rightarrow (\Diamond_{[1.015,1.015]} r_a \wedge \Diamond_{[1.015,1.015]} q_a)$ holds. In the case that the request is granted, the physical train of *TrainA_ Image* just moves on for another 0.7 (T_2) seconds to pass the crossing. In this case, $p_a \rightarrow \Box_{[0,1.7]} \neg r_a$ holds. Therefore, P_v is indeed satisfied by the model. In the next subsection, we will illustrate how to predict real-time property P_r of the realization from P_v, and how to preserve the correctness of the model into the realization (i.e. P is satisfied by the realization).

[9]The weakening or strengthening relation between real-time properties is discussed in (Huang 2005).

```
static PDO *PDM_TurnOff(PDO **LV){ // LV is an object carrying the ID of the target train
  unsigned char trainID; byte buffer[2];
  trainID = (unsigned char) LV[1]→i; // the ID of the target train
  buffer[0]=0x21; // the byte code to turn off a motor
  buffer[1]= 1≪(trainID-1); //converting the ID according to the protocol
  _output(serialPort, buffer[0]); // serial communication
  _output(serialPort, buffer[1]);
  return LV[0];}
```

Fig. 16 A method to implement the interaction of sending a pause message

7.3 System synthesis

In this phase, the model devised during the design stage is automatically transformed into an executable realization. In order to generate a correct realization, two important tasks should be carried out: implementation of communication interfaces and estimation of timing differences.

(1) *Implementation of interfaces:* Synthesis tool Rotalumis can transform a POOSL model into its realization by adopting different strategies to map the process part and the data part of a POOSL model (see Sect. 6.3). For a POOSL model that interacts with the physical world, the standard Rotalumis tool provides the mechanism to convert abstract interactions with the outside world into actual physical interactions. For example, an interaction "sending a *pause* message to the physical environment" (e.g. *a!pause* in *InterfaceA* process) can be implemented by the C++ code given in Fig. 16. For more detailed information about this issue, readers are referred to (van Bokhoven 2002).

(2) *Parameter estimation in the absolute* $[x, y]$*-hypothesis:* In Sect. 6.3, we have demonstrated that the Rotalumis tool supports the absolute $[x, y]$-hypothesis. Parameters x and y are lower and upper bounds respectively of the timing deviation between the observation times of corresponding actions in the model and in the realization. In general, the larger the difference between y and x is, the larger the deviation is between the quantitative real-time properties satisfied by the model and the realization. In practice, we can obtain the values of x and y in various ways. Here we give two examples.

- *Measurement-based approach* During the execution of the realization, the scheduler can record the timing deviation of each action at run time. Then the values of x and y can be estimated according to the recorded timing deviations. Since this approach is based on simulation, it offers the benefit of easy applicability. Furthermore, no estimation is required of the execution time of each action and of the scheduling cost in this case. The major pitfall of this approach is that the analysis results based on simulation techniques may not be reliable, since each simulation run only explores a part of a single trace.
- *Model-based approach* Instead of measuring the parameters of the absolute hypothesis during run-time, we could also use model-based approaches to estimate the parameters. In these approaches, we could make use of existing results in worse-case execution time (WCET) analysis such as (Park 1993; Gupta and Micheli 1997). For instance, we could label each individual action with its WCET. Based on the time labelling, we could do exhaustive analysis of the time bound by

Table 3 Statistics of the timing deviations

Timing deviations (10^{-4} seconds)	0.1–0.5	0.5–1	1–2	2–3	3–5
Frequency	1 877 257	112 813	9909	26	3

analyzing actions to be performed at each virtual time point along each possible execution path. It is also possible to monitor the timing deviations during the simulation of the model (Florescu et al. 2004). However, it is still an interesting topic to investigate how to estimate tight bounds for the absolute hypothesis.

In the scheduler of Rotalumis, an action is executed when its digital hardware time has exceeded its virtual time. In other words, the digital (or analog) hardware time of each action is no earlier than its virtual time. Therefore, we can safely estimate the lower bound x in the absolute $[x, y]$-hypothesis to be 0. In our example, we estimated y using the simulation-based approach.[10] We recorded around 2 000 000 timing deviations of actions between the virtual time and the digital hardware time. To reduce the overhead caused by recording activities, only the timing deviation of the last action observed at each virtual time moment is recorded. As shown in Table 3, most timing deviations fall between 1×10^{-5} and 5×10^{-5} seconds, and only a few timing deviations reach up to 5×10^{-4} seconds.[11] These larger deviations can be contributed to the background activities of the operating system.

We have discussed that the absolute timing differences between the digital hardware time domain and the auxiliary time domain can also contribute to the parameters of the absolute hypothesis. We choose the step-width consisting of 7×10^8 hardware clock steps to construct the auxiliary analog clock (using the derived step-width in Sect. 4.2). Most PC hardware clocks cost less than 1$ and can only provide marginal timekeeping performance. The average error of these clocks is at the order of magnitude 10^{-5} per-second. We assume that the absolute timing difference between the digital hardware clock and the auxiliary analog clock is less than 10^{-3} seconds (about 7×10^5 hardware clock steps).

Based on the above analysis, we estimate that y is 10^{-3}. Therefore, the absolute $[0 - 10^{-3}, 10^{-3} + 10^{-3}]$-hypothesis is complied with. Now, we can predict the quantitative timing properties of the realization in the auxiliary time domain. For example, we already know that property P_v is satisfied by the model in the virtual time domain, so we can predict that property $P_a = R_a^{0.003\Box}(P_v)$ holds in the realization interpreted in the auxiliary time domain. Property P_a is given by the following

$$\Box(p_a \rightarrow ((\Diamond_{[1.012,1.018]} r_a \wedge \Diamond_{[1.012,1.018]} q_a) \vee \Box_{[0,1.712]} \neg r_a)).$$

In this example, the accuracy of the PC hardware clock is sufficient enough. However, in certain situations, to obtain a better preservation result from the auxiliary time

[10]The underlying platform in our case is a PC with a PIII 700 MHz processor, 128 MB memory and the Windows 98 OS.

[11]Note that the timing deviations are observed after the issuing of the actions. Hence, the actual timing deviations should be smaller than this value.

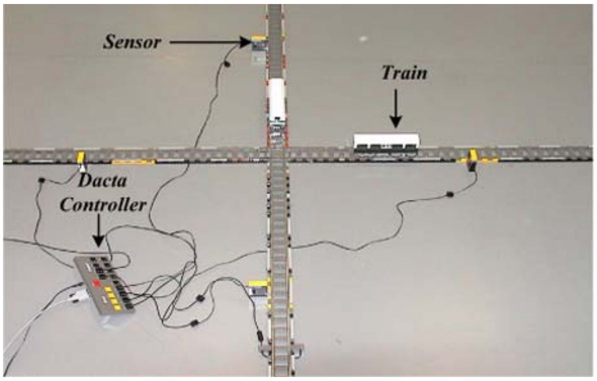


Fig. 17 A snapshot of the realization of the system

domain to the reference time domain, we can replace the low cost clock of the PC with a better oscillator (such as a quartz, a rubidium, or a cesium oscillator), which clock error is in the order of magnitude of 10^{-8} per-second or less (Lombardi 2002).

(3) *Parameter estimation in the relative* $[x, y]$*-hypothesis:* The inaccuracy of the platform clock also affects the quantitative real-time properties of a realization. This influence is addressed by the relative $[x, y]$-hypothesis. In our example, we choose the time of the environment as the reference time, which is counted by a perfect reference clock. Since the average error of most PC hardware clocks is in the order of magnitude of 10^{-5} per-second, we estimate the relative timing drift between the auxiliary analog clock and the reference clock to be 10^{-3}. Then x and y in the hypothesis are 0.999 and 1.001 respectively. Now we can predict that real-time property P_r is satisfied by the realization in the reference time domain. Property $P_r = R_r^{[\frac{1}{1.001}, \frac{1}{0.999}]}(P_a)$ is given by the following formula

$$\Box(p_a \to ((\Diamond_{[\frac{1.012}{1.001}, \frac{1.018}{0.999}]} r_a \wedge \Diamond_{[\frac{1.012}{1.001}, \frac{1.018}{0.999}]} q_a) \vee \Box_{[0, \frac{1.712}{1.001}]} \neg r_a)).$$

Property P_r ensures that after *TrainA_Image* receives the first signal, it either receives a *denied* message and sends out a *pause* message within interval [1.010, 1.020], or continues to move on during time interval [0, 1.711].[12] This is stronger than the required property P in Sect. 7.2. Consequently, we can guarantee that the physical train of *TrainA_Image* never stops inside the crossing area, while still maintaining the operating efficiency of the system. Figure 17 gives a snapshot of a physical realization of the system.

8 Related work

In this paper, we propose a synthesis approach to address the automatic generation of concurrent real-time software from models. Particularly, our approach focuses on

[12]Notice that $[\frac{1.012}{1.001}, \frac{1.018}{0.999}] \subset [1.010, 1.020]$ and $[0, \frac{1.712}{1.001}] \supset [0, 1.711]$, P_r is actually stronger than property $\Box(p_a \to ((\Diamond_{[1.010,1.020]} r_a \wedge \Diamond_{[1.010,1.020]} q_a) \vee \Box_{[0,0,1.711]} \neg r_a))$. For more information about the weakening relation between formulas, readers are referred to (Huang 2005).

minimizing the negative effects of timing differences on the behavior inconsistency between a model and its realization. It also provides a way to predict the quantitative impact of the timing differences between a model and its realization. Compared with the existing synthesis approaches for real-time systems, the proposed approach in this article has the following characteristics.

- The proposed approach gives a systematic way to address the quantitative impact on the correctness of the realization caused by ineliminable timing differences between the model and the realization. Most existing approaches treat the impact of timing differences in a limited way. For instance, the timing differences between the digital hardware clock and the environment clock are not explicitly addressed.
- Most synthesis approaches aim at building a scheduler, which triggers timed processes to satisfy timing constraints of the system (Altisen et al. 2002; Hune et al. 2001) (see also Sect. 8.2 in this section). In the proposed approach, the timing behavior of the model is considered as the "reference" behavior, and the realization is generated as "close" as possible to the model. Based on the distance between the model and the realization, we can predict the properties of the realization and further judge its correctness.
- The proposed approach does not contradict with existing scheduling theory and scheduling algorithms. On the contrary, the combination with the existing scheduling techniques can strengthen the applicability of the approach. For example, in Sect. 9.3, we discuss that real-time properties of interest may be only related with some observable actions. In this case, we measure the distance between the model and the realization based on the timing differences between these observable actions only. The times of unobservable actions do not have a direct impact on the distance measure, but they may have an influence on the times of observable actions, which in turn affect the distance measure. If we consider these unobservable actions as internal actions in the theory of observational equivalence (Milner 1989), the possible change in the execution order between observable actions and unobservable actions does not change the properties of the system, but it may result in a smaller distance (Florescu et al. 2006). In this case, a more sophisticated scheduler is required to manipulate the execution order of all actions.

In literature, many synthesis approaches for real-time software have been proposed. In the following, we compare these approaches with the proposed approach. These approaches are classified into three categories according to their different ways to treat the impact of the timing differences between the model and the realization.

8.1 Digital circuit synchrony hypothesis

One of the related work is in the area of synchronous design approaches. In these approaches, synchronous modeling languages, such as Argos, Esterel, Lustre and Statecharts are used to model reactive systems. The timing semantics of these languages is often based on the *synchrony hypothesis*, which assumes that the underlying machine is infinitely fast, and hence that system reactions are synchronized with the system inputs. The synchrony hypothesis actually makes the same assumption as many formal timed models does. It assumes that actions are instantaneous. The synthesized realization in synchronous design approaches is required to react to any input within exactly one clock cycle, no matter how complex the realization is. In other words, the absolute timing difference between the observation times of actions in the model and in the realization are synchronized within one cycle (which is also called digital circuit synchrony hypothesis (Berry 1992)). Under this restriction, the properties of the realization are considered to be almost the same as those of the model, due to their closeness in distance. The digital circuit synchrony hypothesis can actually be viewed as a special case of the absolute hypothesis.

8.2 WCET-based schedulability analysis

The approaches in this category employ two different models: a communication model and a timed model. In the communication model, actions are assumed to be instantaneous and qualitative properties (such as deadlock, safety and liveness properties) are evaluated based on synchronizations between concurrent components (or processes). In the timed model, concurrent components (or processes) are abstracted by priorities, worst case execution times (WCET) and deadlines. The quantitative real-time properties are guaranteed on the basis of a schedulability analysis. The influence of action execution times in different time domains is usually not explicitly considered and is implicitly assumed to be covered by the WCETs of concurrent processes. Typical examples of these approaches are (Amnell et al. 2002; Henzinger et al. 2003; Liu and Lee 2003).

In (Amnell et al. 2002), the authors use timed automata with a real-time task extension to model real-time systems. A transition of a timed automaton can be associated with the starting of a real-time task, which has parameters such as a priority, a worst case execution time and a deadline. The schedulability of tasks can be checked in the model. During system synthesis, the prefect synchrony hypothesis is employed, which assumes that the execution times of control activities are neglected.

In (Henzinger et al. 2003), the Giotto design methodology based on time triggered architectures is proposed. In a Giotto model, a real-time system consists of a set of concurrent tasks, each of which has a well-defined start and stop time. The communication between tasks can only be carried out at the beginning or at the end of the execution of tasks. These constraints result in a concise framework to reason about the behavior of the system. During system synthesis, the Giotto compiler ensures that timing constraints of tasks are guaranteed by performing a schedulability analysis based on the worse-case execution time of tasks. During run-time, a scheduler invokes each task based on the result of the analysis.

In (Liu and Lee 2003), a framework called Timed multitasking (TM in short) to model real-time software is proposed. Different from the Giotto approach, TM triggers tasks (also called actors) by events instead of by time. Similar to Giotto models, the execution times of tasks are fixed, which also yields a concise framework to reason about the behavior of a system, and the correctness of the realization is also guaranteed by the schedulability analysis.

8.3 Timing difference analysis

In the past years, studying the impact of timing differences between models and realizations has become a promising research area. Till now, there are at least two ways to formally address this problem in literature. One is based on *property preservation* in which one tries to find the most relaxed Realization $R(\epsilon)$ from the model M such that $R(\epsilon)$ can satisfy the desired property P verified in M. The parameter ϵ reflects the extend of the relaxation. Typical work in this area is (Wulf et al. 2005). The other is *property transformation*, which simply predicts properties of the realization R from those of the model M according to their timing difference ϵ. Properties of R are not necessarily the same as those of the model, but the smaller the value of ϵ is, the closer their properties are. Typical work in this scope is (Huang et al. 2003; Henzinger et al. 2005; Huang et al. 2006). Note that in this scope, to ensure that R satisfies a property P in the end, we can require that M satisfies a stronger property P' so that the relaxation of P' is still stronger than P. In the following, we give a more detailed comparison between the property-preservation technique (Wulf et al. 2005) and the property-transformation technique (Huang et al. 2003).

In (Wulf et al. 2005), the authors address the differences in timing semantics between a model and its realization. In their work, models of real-time systems are represented by timed automata. The timing semantics (called Almost ASAP) of their corresponding realizations can be derived from the relaxation of timing constraints (or guards) in timed automata. There are several differences between the work in (Wulf et al. 2005) and our work. An essential one is the different assumptions that are used to relax the timing semantics of the model in the realization. This is illustrated by the following example. Assume that the model has a timed trace $\bar{\tau}$ given by $(\alpha_1, t_1), (\alpha_2, t_2) \ldots (\alpha_i, t_i) \ldots$. According to the relaxation mechanism proposed in (Wulf et al. 2005), the following set of relaxed timed traces $\{\bar{\tau}' \mid \forall i \in N, |t_i - t_{i-1} - (t'_i - t'_{i-1})| \leq \epsilon \wedge \alpha'_i = \alpha_i\}$ are valid in the realization with relaxation parameter ϵ. In contrast, according to the relaxation mechanism proposed in our work, a set of relaxed timed traces $\{\bar{\tau}' \mid \forall i \in N, |t_i - t'_i| \leq \epsilon \wedge \alpha'_i = \alpha_i\}$ can be observed in the realization which is absolute $[-\epsilon, \epsilon]$-close to the model. Intuitively speaking, the former one relaxes the timing constraints based on the "relative" timing relation between two adjacent actions in the sequence, while the latter one relaxes timing constraints based on the "global" time of actions in the sequence.

In our work we go further than the theoretical work described above. First we have carefully constructed intermediate time domains to reduce the timing differences between the model and the realization. In addition, a synthesis approach has been developed to guide software synthesis. We have also implemented the approach in a concrete tool set.

In this section, we mainly focused on the approaches which generate real-time realizations from the model. It should be noticed that there is a lot of other work carried out to solve an inverse problem on how to make an adequate model for an existing system to analyze its real-time properties. For instance, in (Gupta et al. 1997; Puri 2000; Alur et al. 2005), the authors address the creation of models that are robust w.r.t. timing deviations.

9 Open issues and future work

In this article, we have introduced a synthesis approach to automatically generate reliable real-time software from models. It is most suitable for synthesizing concurrent control systems running on a single processor platform, where a central scheduler ensures the absolute closeness between a model and its realization in the digital hardware time domain. In this section, we will further investigate some possible extensions of this work and existing open issues.

9.1 Distributed applications

In a distributed system, the software components are deployed on multi-processor platforms using (different) hardware clocks. The proposed work in this article can be applied to the distributed system in the following way: software components on the same processor are synthesized independently from the components on other processors. Although each synthesized sub-system runs on a different processor with a different hardware clock, its real-time properties in a unified reference time domain can be predicted by using the proposed approach. Consequently, the real-time properties of the whole system can be analyzed on the basis of real-time properties of sub-systems in the unified reference time domain. However, this approach can not be applied to directly predict global properties of the whole system, since no central scheduler exists to ensure the compliance of the absolute $[x, y]$-hypothesis for the whole system. Even when a central scheduler is implemented in the distributed environment, it would possibly be inefficient to ensure the compliance of the absolute $[x, y]$-hypothesis due to the communication overhead among distributed processors. Further investigation is still needed for automatically synthesizing distributed systems in a correctness-preserving way.

9.2 Computationally intensive applications

For computationally intensive applications such as video- and image- processing applications, the execution of some actions may take a significant amount of physical time, which may lead to a large interval $[x, y]$ in the absolute $[x, y]$-hypothesis. As a consequence, the quantitative real-time properties predicted in the realization can be far away from those in the model. However, this problem can be relieved by introducing a proper abstraction mechanism, which hides computationally intensive actions. For example, in process calculus (e.g. CCS and CSP), an abstraction mechanism is used to hide actions that are not observable or not interesting, and certain properties of the system can be verified based on observable actions alone. Consequently, the transformation of these properties is determined by the deviation bounds between the observation times of observable actions (Florescu et al. 2006).

9.3 Parameter estimation of the absolute $[x, y]$-hypothesis

A key issue of our approach is the estimation of the parameters in the absolute $[x, y]$-hypothesis. The estimation results give feedback to designers to further improve models or ensure the correctness of realizations. In Sect. 7.3, we have given a brief discussion of possible measurement-based and model-based estimation approaches. It is desirable to obtain the bounds of interval $[x, y]$ as tight as possible.

For large systems, the accurate estimation of the parameter values is not a trivial task. For instance, in the measurement-based approaches and the model-based approaches based on simulation, the estimation process only covers a small fraction of the possible traces. Some traces, which occur with a probability, may not be covered by the estimation process and may have a larger deviation when they are executed in the realization. Consequently, the estimation results may not be reliable. In the approaches based on the exhaustive technique, the estimation process often suffers from the well-known state-space explosion problem. However, it has been theoretically proven that it is possible to compute the distance between two timed systems using an EXPTIME algorithm in (Henzinger et al. 2005). The distance metric used in (Henzinger et al. 2005) is consistent with the absolute metric in this article. Therefore, their algorithm also provides a way to compute the parameters of the absolute hypothesis.

10 Conclusions

In this article, we first used an example to show that functional inconsistencies can be observed during the synthesis of a real-time system from its model. These inconsistencies can be contributed to the large time gap between the model in the virtual time and the realization in the reference time. To bridge this gap, we built a series of dedicated intermediate time domains between the virtual time domain and the reference time domain (see Fig. 8). Two parameterized hypotheses (the absolute and relative $[x, y]$-hypotheses) were proposed to capture bounds of the absolute timing differences and the relative timing differences between various time domains. The satisfaction of the two hypotheses ensures that qualitative real-time properties can be preserved between different time domains. Furthermore, the parameters of both hypotheses can be used to predict quantitatively the properties of the realization. Following that, we proposed a tool set for real-time software synthesis, which complies with the absolute $[x, y]$-hypothesis. Finally, we used the proposed synthesis approach to design a railroad crossing system. The correctness-preservation of software synthesis is illustrated by preserving an important quantitative property from the model to the realization in the reference time domain.

References

Altisen K, Gossler G, Sifakis J (2002) Scheduler modeling based on the controller synthesis paradigm. Real-Time Syst 23(1–2):55–84

Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235

Alur R, Torre SL, Madhusudan P (2005) Perturbed timed automata. In: Proceedings of eighth international workshop on hybrid systems: computation and control

Amnell T, Fersman E, Pettersson P, Yi W, Sun H (2002) Code synthesis for timed automata. Nord J Comput 9(4):269–300

Amnell T, Fersman E, Mokrushin L, Pettersson P, Yi W (2003) TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Formal modeling and analysis of timed systems: first international workshop, FORMATS 2003. Springer, Berlin, pp 60–72

Berry G (1992) A hardware implementation of pure Esterel. In: Academy proceedings in engineering sciences, vol 17. Indian Academy of Sciences, pp 95–130

Florescu O, Voeten J, Huang J, Corporaal H (2004) Error estimation in model-driven development for real-time software. In: Proceedings of forum on specification and design language, FDL'04, Lille, France

Florescu O, Huang J, Voeten J, Corporaal H (2006) Strengthening property preservation in concurrent real-time systems. In: Proceedings of the IEEE international conference on embedded and real-time computing systems and applications (RTCSA), Sydney, Australia, pp 106–109

Geilen M (2002) Formal techniques for verification of complex real-time systems. PhD thesis, Eindhoven University of Technology, The Netherlands

Geilen M, Voeten J, van der Putten P, van Bokhoven L, Stevens M (2001) Object-oriented modelling and specification using SHE. J Comput Lang 27:19–38

Glasser U (1998) ASM semantics of SDL: concepts, methods, tools. In: 1st SAM workshop on SDL and MSC, pp 271–280

Glasser U, Gotzhein R, Prinz A (2003) The formal semantics of SDL-2000: status and perspectives. Comput Netw Int J Comput Telecommun Netw 42(3):343–358

Gupta R, Micheli GD (1997) Specification and analysis of timing constraints for embedded systems. IEEE Trans Comput Des Integr Circuits Syst 16(3):240–256

Gupta V, Henzinger T, Jagadeesan R (1997) Robust timed automata. In: Maler O (ed) Hybrid and real-time systems, proceedings of international workshop HART'97, Grenoble, France. Springer, Berlin, pp 331–345

Heitmeyer CL, Jeffords RD, Labaw BG (1993) A benchmark for comparing different approaches for specifying and verifying real-time systems. In: Proceedings of the tenth international workshop on real-time operating systems and software

Henzinger TA, Manna Z, Pnueli A (1992) Timed transition systems. In: Proceedings of the real-time: theory in practice, REX workshop, London, UK, Springer, Berlin, pp 226–251

Henzinger T, Kirsch C, Sanvido M, Pree W (2003) From control models to real-time code using Giotto. IEEE Control Syst Mag 23(1):50–64

Henzinger TA, Majumdar R, Prabhu V (2005) Quantifying similarities between timed systems. In: Proceedings of the third international conference on formal modeling and analysis of timed systems (FORMATS). Lecture notes in computer science, vol 3829. Springer, New York, pp 226–241

Huang J (2005) Predictability in real-time system design. PhD thesis, Eindhoven University of Technology, The Netherlands

Huang J, Voeten J, Geilen M (2003) Real-time property preservation in approximations of timed systems. In: Proceedings of 1st ACM and IEEE international conference on formal methods and models for codesign. IEEE Computer Society, Los Alamitos, pp 163–171

Huang J, Geilen M, Voeten J, Corporaal H (2006) Branching-time property preservation between real-time systems. In: Proceedings of fourth international symposium on automated technology for verification and analysis 2006, Beijing, Springer, Berlin, pp 260–275

Hune T, Larsen K, Pettersson P (2001) Guided synthesis of control programs using UPPAAL. Nord J Comput 8(1):43–64

Koymans R (1990) Specifying real-time properties with metric temporal logic. Real-Time Syst 2(4):255–299

Larsen KG, Pettersson P, Yi W (1997) UPPAAL in a nutshell. Int J Softw Tools Technol Transf 1(1–2):134–152

Liu J, Lee E (2003) Timed multitasking for real-time embedded software. IEEE Control Syst Mag 23(1):65–75 (special issue on Advances in software enabled control)

Lombardi M (2002) Time and frequency from a to z. http://tf.nist.gov/general/glossary.htm

Milner R (1989) Communication and concurrency. Prentice Hall, New York, ISBN 0-13-114984-9

Newton I (1999/1687) The principia: mathematical principles of natural philosophy. University of California Press, Berkeley (edited by I. Bernard Cohen and Anne Miller Whitman)

Nicola RD, Vaandrager F (1990) Action versus state based logics for transition systems. In: Proceedings of the LITP spring school on theoretical computer science on semantics of systems of concurrent processes, pp 407–419

Nicollin X, Sifakis J (1994) The algebra of timed processes, ATP: theory and application. Inf Comput 114(1):131–178

Noonan L, Flanagan C (2004) Modeling a network processor using object oriented techniques. In: Proceedings of the digital system design, EUROMICRO systems on (DSD'04), Washington, DC. IEEE Computer Society, Los Alamitos, pp 484–490

Park CY (1993) Predicting program execution times by analyzing static and dynamic program paths. Real-Time Syst 5(1):31–62

Puri A (2000) Dynamical properties of timed automata. Discret Event Dyn Syst 10(1–2):87–113

Smyth N (1998) Communicating sequential processes domain in Ptolemy II. MS Report UCB/ERL Memorandum M98/70, Dept. of EECS, University of California, Berkeley

Stotts PD, Pratt T (1985) Hierarchical modeling of software systems with timed petri nets. In: International workshop on timed petri nets. IEEE Computer Society, Los Alamitos, pp 32–39

Theelen B, Voeten J, van Bokhoven L, van der Putten P, de Jong G, Niemegeers A (2001) Performance modeling in the large: a case study. In: Proceedings of the European simulation symposium

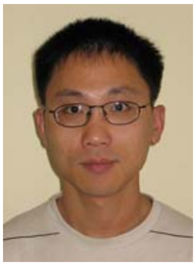
Theelen B, Voeten J, Kramer R (2003) Performance modelling of a network processor using POOSL. J Comput Netw 41(5):667–684 (special issue on Network processors)

van Bokhoven L (2002) Constructive tool design for formal languages from semantics to executing models. PhD thesis, Eindhoven University of Technology, The Netherlands

van der Putten P, Voeten J (1997) Specification of reactive hardware/software systems. PhD thesis, Eindhoven University of Technology, The Netherlands

van Wijk F, Voeten J, ten Berg A (2003) An abstract modeling approach towards system-level design-space exploration. In: System specification and design languages. Kluwer Academic, Dordrecht, pp 267–282

Wulf MD, Doyen L, Raskin J-F (2005) Almost asap semantics: from timed models to timed implementations. Formal Aspects Comput 17(3):319–341



Jinfeng Huang received his B.Eng. in computer science from China University of Mining and Technology, China, in 1997, and his M.Sc. in computer science from Xi’an Jiaotong University, China, in 2000 respectively. In 2005, he received his Ph.D degree from Eindhoven University of Technology, The Netherlands, for his work on predictable real-time software design. Since April 2005, he works as a post-doc researcher at Electrical Engineering Department, Eindhoven University of Technology. His research interests include formal methods on concurrent, real-time and distributed systems and software synthesis.

Jeroen Voeten received his master’s degree in Mathematics and Computing Science in 1991 and his Ph.D. in Electrical Engineering in 1997 from the Eindhoven University of Technology, the Netherlands. Since 1997 he is working as an assistant professor in the Electronic Systems group at the faculty of Electrical Engineering. As from January 2005 he is also working as a senior research fellow at the Embedded Systems Institute in Eindhoven. His research interests include system-level design methodology and performance modeling for embedded systems.

Henk Corporaal has gained a MSc in Theoretical Physics from the University of Groningen, and a PhD in Electrical Engineering, in the area of Computer Architecture, from Delft University of Technology. Corporaal has been teaching at several schools for higher education, worked at the Delft University of Technology in the field of computer architecture and code generation, had a joint appointment at the National University of Singapore, has been scientific director of the joined NUS-TUE Design Technology Institute, and has been department head and chief scientist within the DESICS (Design Technology for Integrated Information and Communication Systems) division at IMEC, Leuven (Belgium). Currently Corporaal is Professor in Embedded System Architectures at the Einhoven University of Technology (TU/e) in The Netherlands. He has co-authored many papers in the (multi-)processor architecture and embedded system design area. Furthermore he has invented a new class of VLIW architectures, the Transport Triggered Architectures; a book about these architectures has been published. His current research projects are on the predictable design of soft- and hard real-time embedded systems.