Predicting implementation accuracy for real-time control systems

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Predicting Implementation Accuracy for Real-Time Control Systems

Oana Florescu, Jeroen Voeten, Henk Corporaal

Predicting implementation accuracy for real-time control systems

Oana Florescu, Jeroen Voeten, Henk Corporaal

Eindhoven University of Technology
Electrical Engineering Department
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
o.florescu@tue.nl

Abstract

Model-driven approaches proved themselves not suited yet to support real-time software development. Even if they have the ability of capturing adequately both functional and non-functional (timing) characteristics of a system, they still lack an appropriate mechanism of generating an implementation from a model while preserving the properties verified. In previous work we have proven that, if the implementation trace is very close (ϵ-close) to a model trace, the properties verified in the model are preserved up to ϵ in the system realisation. This deviation is due to the model assumption of zero-time for computational actions that, in reality, no target platform can ensure. This paper proposes an approach for estimating the time-deviation between model and implementation, by modelling the realisation of the system when software components would run on the target platform. The approach is based on Software/Hardware Engineering method for complex real-time systems design and the Y-chart scheme concepts.

1 Introduction

Complex embedded real-time systems are often comprised of a combination of many hardware and software components that are supposed to synchronise and coordinate different processes and activities. From early stages of the design, many decisions must be made to guarantee that the realisation of such a complex machine meets all the functional and non-functional (timing) requirements. The more inappropriate design decisions are made, the more the time to develop the new product increases and often exceeds the technology innovation cycle. Therefore, by the time the new product is released on the market, it may already be outdated or it could be produced at a lower cost due to some newly discovered technology. For these reasons, the industry focus had to shift from the improving of the system implementation itself to the improving of the efficiency of the design process and raising the quality of the engineered product. To cope with this, the introduction of tools for hardware and software design, the wider use of performance analysis methods, the adoption of the Unified Modelling Language (UML) [12] and the emerging of the Model-Driven Architecture (MDA) [11] concepts are a few examples of attempts.

Due to the intrinsic characteristics of embedded real-time systems, from early stages of the design, engineers need to understand the properties of the system, whether these will meet or not the overall functional and non-functional (timing) requirements. Moreover, they need to guarantee that these properties will be satisfied by the implementation of the systems as well as the model. As it was shown in [3], an appropriate model-driven design methodology should provide: (i) a suitable modelling technique that can appropriately capture functional and timing properties in models to reason on a well-founded basis about them, and (ii) a model synthesis mechanism that can generate the realisation of the model while preserving the properties analysed.

1

1.1   Related work

The Object Management Group (OMG) has adopted the Unified Modelling Language (UML) [12] as a standard facility for constructing models of object-oriented software. UML has proven to be suitable to model qualitative aspects of a system and there were extensions defined to it to provide a standardised way of denoting timing aspects for real-time systems [13]. Nevertheless, application of mathematical analysis techniques remains complicated due to the difficulty of relating formal techniques to UML diagrams. Although UML is largely used both in industry and academia, it has shown not to be very appropriate for modelling hard real-time systems as it lacks a formal semantics, restraining it from a proper analysis of system behaviour.

Based on UML, a couple of design approaches to support model-driven development for real-time systems were conceived. However not all of them satisfy all the characteristics of hard real-time design, namely to have both a well-founded and expressive modelling language as well as an automatic and correctness-preserving transformation technique.

Real-Time Object-Oriented Modelling (ROOM) method [14], based on UML, showed itself not suitable for modelling real-time systems due to ROOM's platform-dependent semantics. ROOM relies on an asynchronous timing mechanism that refers to the physical clock. Therefore, the results of a model simulation are dependent on the target hardware. Consequently, accurate analysis and prediction of timing behaviour of a system cannot be made [6]. Regarding the automatic code generation, the implementation of a software component is obtained by linking the model to a service library, which actually acts like a virtual machine on top of the target platform. The implementation is in fact an executable model, so it inherits all the problems related to the platform-dependent semantics encountered in the model. Besides the lack of a proper model analysis, the properties specified in the model cannot be satisfied by the implementation.

Another UML-related design approach, TAU2 [15] relies on the concept of virtual time whose progress is not directly affected by the progress of the physical time. In this way, real-time behaviour of models is well-defined in a platform-independent way. Although TAU2 can provide a reliable way of analyzing a model it does not have a reliable transformation mechanism to guarantee preservation of model-verified properties in the implementation. During automatic code generation, model time is replaced by physical time and all expressions referring to some amount of time will refer to *at least* that amount of time. As timing errors are accumulated during the execution, the issuing moment of different actions can deviate from those observed in the model and also the ordering of the events can change, so model and implementation are not consistent [6].
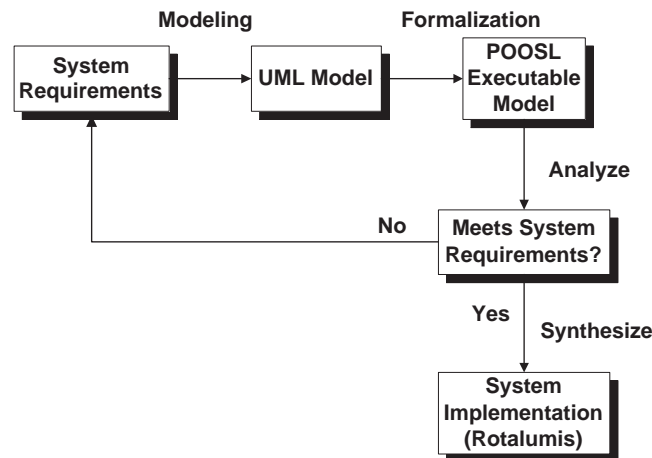

Figure 1. SHE method for real-time systems design

To support the design and development of real-time systems, Software/Hardware Engineering (SHE) method [17]

2

provides a well-founded and expressive language, Parallel Object-Oriented Specification Language (POOSL) [17], for modelling and analysis of complex real-time systems, and a correctness-preserving transformation tool (Rotalumis [16]) for model synthesis. SHE uses a UML profile to formulate the concepts needed for the realisation of the requested functionality of a system. POOSL formalises the behaviour specified in informal UML diagrams, establishing a formal model. The mathematical semantics of the language enables formal analysis of the model; moreover, automatic generation of implementation that preserves model properties is possible.

1.2 Contributions

A model represents an approximation of a system realisation because it removes or hides the irrelevant details to allow designers focus more easily on the essentials with the purpose of analysing the system properties. However, due to all the approximations made in a model, the real-time properties verified cannot hold precisely in the software implementation. This report proposes an approach to estimate the size of the time-deviation between a model and its implementation on a certain platform. This approach enables the prediction of how accurate the implementation of the system can be on a certain target. This prediction would help in making trade-offs between different architectures and the accuracy of the control software and would offer hints for possible re-designs of the system in order to cover for the time-deviation.

The paper is organised as follows. Section 2 presents the semantics of POOSL language, the notion of real-time properties preservation and how the code generation is performed. This gives the necessary background information needed to understand the problem and the need for time-deviation predictions. The approach for determining this deviation is described in Section 3. A case study is given in Section 4 and conclusions are drawn in Section 5.

2 Preliminaries

The Parallel Object-Oriented Specification Language (POOSL) [17] lies at the core of the SHE design method. The semantics of POOSL models is presented in 2.1. Moreover, the real-time properties, discussed in 2.2, can be preserved in the synthesis approach, as shown in 2.3.

2.1 POOSL semantics

POOSL is equipped with mathematical semantics that can formally describe concurrency, distribution, synchronous communication, timing and functional features of a system in a single executable model, using a small set of very powerful primitives. Primitives can be combined in an unrestricted fashion and any combination has a precisely defined meaning. The formal semantics guarantees a unique and unambiguous interpretation of a POOSL model, guided by semantical axioms and rules, independent of the underlying execution platform. In [3], we investigate the importance of the formal semantics of a modelling language in supporting the predictability of the system design process.

POOSL consists of a process part and a data part. The process part (processes and clusters), based on a real-time extension of the process algebra CCS (Calculus of Communicating Systems) [9], is used to specify the real-time behaviour of active components. The data part, based upon traditional concepts of sequential object-oriented programming, is used to specify the information that is generated, exchanged, interpreted or modified by the active components.

The semantics of POOSL is defined as a timed labelled transition system which is a mathematical structure that considers a system as an entity having some internal state and, depending on that state, it can engage in transitions leading to other states. Such a transition might be autonomous or stimulated by the environment.
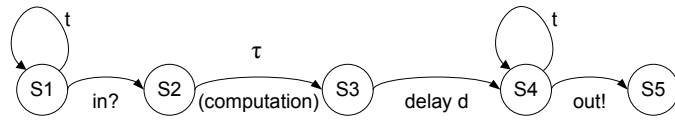
3

Figure 2. Timed labelled transition system

A timed labelled transition system (TLTS) is a graph, as the example in fig. 2, consisting of nodes (states) and labelled edges (transitions). The label represents either the observable behaviour associated with the transition or the time that passes by until the system can engage in another observable activity.

```
in?input(x);              /* x is received */
par
  computation(x)(y)   /* x is the input, y is the output */
and
  delay d
rap;
out!output(y);           /* y is sent */
```

Figure 3. Example of POOSL model

Formally, a timed labelled transition system is a 5-tuple $(S, Act, T, \rightarrow_{Act}, \rightarrow_{T+})$ consisting of a set S of states, a set Act of *actions*, a time domain T, an action transition relation $\rightarrow_{Act} \subseteq S \times Act \times S$ and a time transition relation $\rightarrow_{T+} \subseteq S \times T^+ \times S$, as it is defined in [2]. An action transition $(s_1, a, s_2) \in \rightarrow_{Act}$, also written as $s_1 \rightarrow^a_{Act} s_2$, denotes a possible transition of the system from state s_1 to state s_2 by exchanging action a with its environment. At times, it is convenient to discriminate certain transitions that are considered unobservable. The TLTS has a special label that denotes the internal unobservable transitions, τ, which can also be seen in fig. 2.
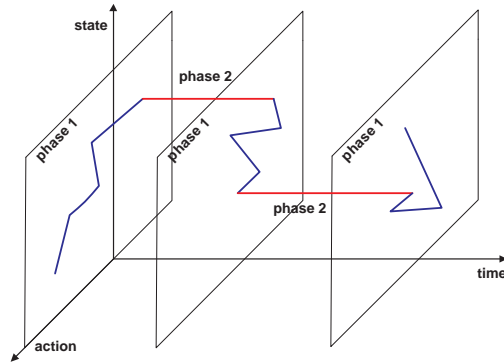
Figure 4. Two phases of the execution

In this kind of model for timed systems, action transitions ($\rightarrow_{Act}$) are assumed to be instantaneous (taking no time), and the passage of an amount of time t is denoted by a transition $s_1 \rightarrow^t_{T+} s_2$. Therefore, the execution has two phases [10], as shown in fig. 4: the state of a system changes either by asynchronously executing atomic actions, such as communication or data computation, without passage of time (phase 1), or by letting time pass synchronously without any action being performed (phase 2). This model assumes an abstract notion of time, called *virtual* or *model time*. The virtual time represents a parameter of the system which is incremented, when a time transition is taken, with its value.

4

By adopting the TLTS for modelling reactive systems, the behaviour of a system can be considered as the set of all sequences of actions observed along any path in the transition system. Each action is observed at a certain moment in time. Therefore, the behaviour of the system is a set of *timed* action sequences.

Let us look at the simple POOSL model example given in fig. 3. Its timed labelled transition system representation is shown in fig. 2, where `in?` and `out!` represent the abbreviated names of the communication actions specified in the model and `computation` is an internal unobservable action. According to the formal semantics of the language, at some moment in time, the input data `x` is received from the environment and the `computation` is performed with this data. After the passage of an amount of `d` units of time, the result is sent to the environment. In fig. 5, a possible timed action sequence of the system is shown.
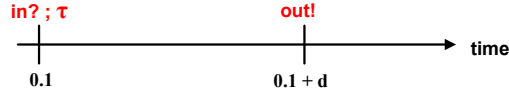

Figure 5. A timed action sequence

2.2 Properties and property-approximation

The goal of the analysis of a real-time system is the verification of certain properties - what actions (events) happen and when. The model of a system has a particular real-time property (e.g. that a certain event happens at a particular moment in time) if and only if all the timed action sequences of the model satisfy that real-time property.

The real-time properties are often formalised using temporal logics, a popular framework able to express observable features and to relate them over time. For this purpose, we use $MITL_\Re$ [4] logic, which is a minor extension of $MITL$ [1].

An example of a real-time property for the system specified in the previous subsection is

$$p \rightarrow \diamond_{[d,d]} q \tag{1}$$

where p is the property that the communication action `in?input(x)` happens, q is the property that the communication action `out!output(y)` happens, $\diamond_{[d,d]}$ means that, *eventually*, after p becomes true at some moment in time, q becomes true within the time interval $[d,d]$, which in fact contains a single point in time. Assuming that the environment is always ready to receive data from the system, then, eventually, the input data is read and after `d` units of time, the output data is written. Note that, when all possible timed action sequences of a system satisfy a real-time property written as a $MITL_\Re$ formula, the timed labelled transition system also satisfies that formula.

Furthermore, a notion of weakening of formulas is defined in [4]. For example, a 0.1-weakening of formula 1 would be

$$p \rightarrow \diamond_{[d-0.1,d+0.1]} q \tag{2}$$

whereas a 0.9-weakening of it would be

$$p \rightarrow \diamond_{[d-0.9,d+0.9]} q \tag{3}$$

Formula 3 is weaker than 2 with respect to formula 1. If a timed action sequence satisfies a real-time property expressed as a $MITL_\Re$ formula, then it also satisfies an ϵ-weakening of it [4]. The larger the value of ϵ, the more weakened the real-time property is.

For two timed action sequences that share the same sequence of actions, a notion of distance is defined in [4]. The metric is based on the time-deviation between the moments in time when the corresponding actions in the two sequences happen. The distance between them is the least upper bound of the absolute difference between

5

the corresponding execution moments of an action. As an example, fig. 6 shows two action sequences that have the same sequence of actions: in?, τ, out!. As it can be seen, there is a time-deviation between the moments in time when in? happens in the two action sequences. Similarly, there are time-deviations for τ and for out! respectively. Because there is a finite number of actions, the distance between these two timed action sequences is given by the maximum time-deviation, which is the one for τ.
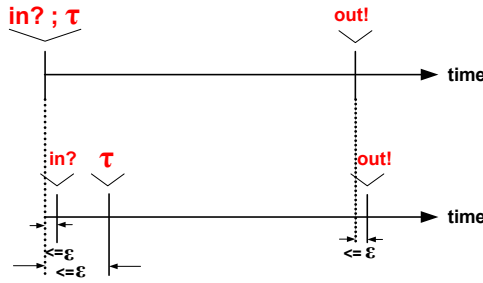

Figure 6. Two finite timed action sequences ϵ-close

In [4], it was also proved that, if two action sequences are ϵ-close (the distance between them is smaller than some ϵ) and a certain real-time property is satisfied by one of the sequences, the other one satisfies a ϵ-weakening of it. In other words, we can say that the second action sequence satisfies the same real-time property up to ϵ. This mathematical result can be applied to prove that the realisation of a system preserves the properties verified in the model, which is discussed further.

2.3 Property-preserving code generation

In this paper, both model and implementation of a system are viewed as sets of timed action sequences. The code generation tool, Rotalumis [16], tries to obtain an implementation of a system consistent with its POOSL model by satisfying the so-called ϵ-hypothesis [3]. The ϵ-hypothesis states that for every timed action sequence in the implementation there is an action sequence in the model that is ϵ-close. In case the ϵ-hypothesis is complied with during the code generation, the implementation preserves *all* the properties of the model up to ϵ.

To satisfy the hypothesis, Rotalumis applies two techniques: it generates a timed action sequence that is part of the set of timed action sequences of the model; moreover, it tries to keep a time-deviation of maximum ϵ between the generated action sequence and its corresponding one in the model.

The algorithm for generating a timed action sequence from a POOSL model has been proven correct with respect to the formal semantics of the language in [2]. According to this algorithm, the data part of a model is directly translated into corresponding C++ expressions, whereas the process part is automatically translated into Process Execution Trees (PETs) (see fig. 7a and fig. 7b respectively). A PET represents the remaining behaviour of a POOSL process. The leaves of the tree are statements describing the timed behaviour of that process, whereas the internal nodes represent compositions of their children (e.g. parallel, sequential, nondeterministic choice). As presented in subsection 2.1, there are two phases during execution. First, a PET scheduler asynchronously grants all the eligible atomic actions, such as communications or data computations, which take no model time, until there are no other actions available at the current virtual time moment. The internal state of each PET changes according to the choices made by the scheduler. Then, according to the semantics of the model, time should pass synchronously for all PETs until the moment when an action becomes eligible again. However, in *physical time*, the execution of actions takes time, thus physical time progresses while virtual time does not. Therefore, there appears the time-deviation between model and implementation. In order to avoid the accumulation of such time-deviation, when a time transition is to be taken, the virtual time progresses as specified in the model, while,
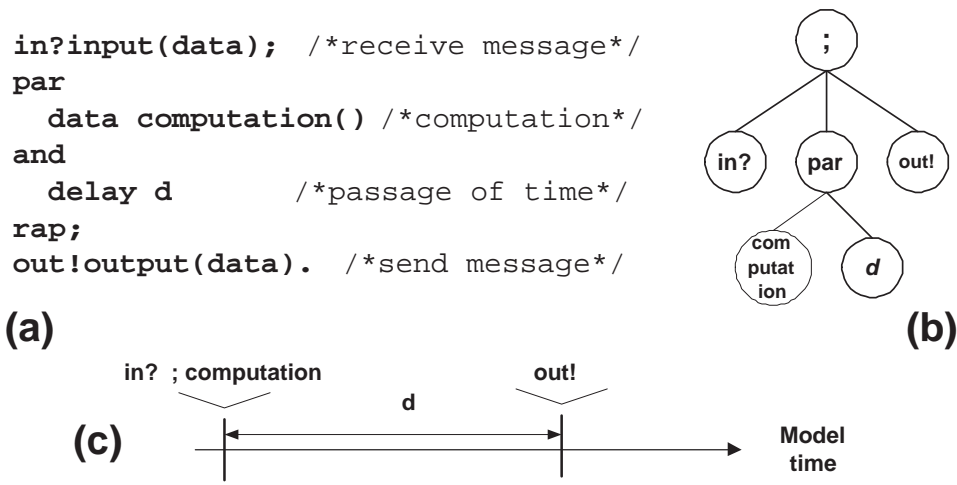
6

```
in?input(data);  /*receive message*/
par
  data computation() /*computation*/
and
  delay d         /*passage of time*/
rap;
out!output(data).  /*send message*/
```

(a)

(b)

(c)

Figure 7. POOSL model example. (a) A POOSL process; (b) Its PET; (c) Its model timed trace

in physical time, the system lets time passes until the moment corresponding to the virtual time of the model is reached. Afterwards, the first phase of the execution is resumed.

For analysis purposed, Rotalumis keeps the value of the time-deviation obtained together with the timed trace represented in model time (fig. 7c). As a result, the synthesised realisation exhibits the same behaviour as the model (i.e. it has the same sequence of actions as the model has). However, the time-deviation induces a weakening of the real-time properties of the model. Due to the mechanism of avoiding the accumulation of the time-deviation, this weakening is bounded to some ϵ value, thus it can be guaranteed that *all* real-time properties of the model are preserved up to ϵ.

3 Time-deviation predictions from models

As it was shown in the previous section, the synthesis mechanism can ensure the preservation of properties up-to a time-deviation. The smaller this deviation, the more accurate the implementation of the system is. Therefore, it is important to predict from the model what the size of this deviation would be. Based on this prediction, changes like choosing a higher performance platform or re-designing the software part of the model such that a large deviation can be accommodated can be made.

In the following, we will discuss the Y-chart scheme concepts in 3.1 on which the construction of a model for analysis is based. Then, we will present how the various parts of the Y-chart components are modelled (in 3.2, 3.3, 3.4 and 3.5) and analysed (in 3.6) in order to obtain the value of the time-deviation.

3.1 Y-chart Concepts

The Y-chart scheme (fig. 8), introduced in [7], is used for design-space exploration. This scheme makes a distinction between applications (the required functional behaviour) and platforms (the infrastructure used to perform this functional behaviour). The design space can be explored by evaluating different mappings of applications onto platforms.

A functional model is specified in terms of a collection of communicating tasks and a resource model is specified as a collection of (parameterised) computation, communication and/or storage resources that are capable of executing the desired functional behaviour [18]. With parameterised resource entities it is easier to check different architecture combinations.
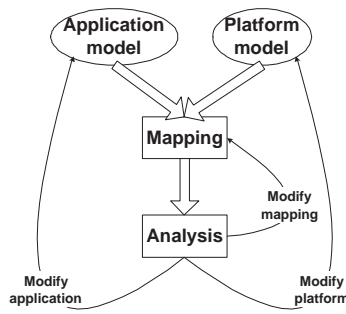
7

Figure 8. Y-chart scheme

In traditional approaches, the separation of system desired behaviour from the description of the underlying platform implies that timing behaviour comes into sight only when the mapping is done. Before the mapping is done, the application model describes how the system should behave and the platform model specifies what it is able to process and under what restrictions (regarding time, storage amount, communication). Tasks specify the desired behaviour in terms of task-level instructions, while resources have to process tasks requests in order to accomplish this behaviour. Mapping an application on a platform means to map each entity in the functional model (task) onto a resource entity in the resource model (CPU). At the task level, the amount of time needed for processing is not known until the mapping stage, because this information is comprised in the description of resource entities. An accurate analysis regarding timing behaviour of the system can only be made after the mapping is done, as the model time elapses only in resources. We can say that the application model (functional behaviour) is influenced by the platform model whose presence is mandatory in order to make reasonable predictions about non-functional system properties. By considering only the model of the application, one cannot reason properly about timing aspects, which is indispensable for real-time systems.
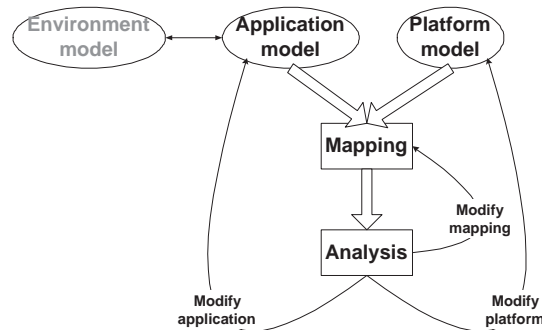

Figure 9. Y-chart scheme updated

As already shown in the previous sections, when modelling real-time systems it is very important to capture both functional and timing requirements in order to make a proper analysis and to automatically generate a software implementation that is consistent with the model. Therefore, in the application model the timing behaviour must be comprised as well, such that it is possible to analyse the timing behaviour of the system from the application model, *without involving the underlying architecture*. Moreover, it is also required to model the *environment* that is controlled by the software. The environment is physically coupled to the platform (by input/output pins), but logically to the application. Therefore, we have updated the Y-chart to look like in fig. 9. Only tasks will be mapped on the platform model, but the model of the controlled environment is needed for reasoning about the system behaviour as a whole. The hardware platform, on which the software will run, is required to support the execution of the application to achieve both functionality and timing behaviour specified in the model. The mapping of the

application on the platform represents the assignment of each software component to a computational resource that mimics the execution of its requests by time delays. The main change from the previous scheme is that the mapping in this case *does not influence* the behaviour of the software model: the platform is *notified* by the software components (tasks) about the actions they *have done* and that in the real system *it* should perform. We call this mapping a model of a *realisation* because it gives a picture of how the system in reality would run on a particular platform configuration. Different mappings on different architectures can be analysed in order to find the most suitable one. The suitability of a mapping is analysed in terms of the size of the timing-deviation (ϵ) introduced. The ϵ indicates to what extent the platform can satisfy the timing requirements of the system. In the following sections the way a model is made based on the updated Y-chart is described.

3.2 Application model

The application part of a real-time system is modelled with POOSL process objects. Using the primitive of the language, any kind of real-time behaviour can be expressed.

As taks may communicate with each other, the communication is modelled by messages exchanged through POOSL channels (`crossing!accessRequest`). An example of a task behaviour modelled in POOSL is given in fig. 10a. In order to model the behaviour of the system when the application is mapped on the platform, in the application model, after each action that would be performed on a processor, a message is sent to the platform to notify a certain activity (`mproc!execute(`action name`)`) (as shown in fig. 10b). In the coming subsection, it will be explained how the platform model is modelled to receive anytime messages from the application model to avoid the influence on the timing behaviour of the application. When such a message arrives at the platform, a *Request* POOSL data object is built, containing the release time, the name and the execution time of the action to be executed.

```
GETACCESSTOCROSSING()()          GETACCESSTOCROSSING()()
  crossing!accessRequest;          crossing!accessRequest;
                                   mproc!execute(msgSent);
  sel                              sel
    crossing?accessDenied;           crossing?accessDenied;
                                     mproc!execute(msgRecv);
    train!stop;                      train!stop;
                                     mproc!execute(msgSent);
    crossing?accessGranted;          crossing?accessGranted;
                                     mproc!execute(msgRecv);
  or                               or
    crossing?accessGranted;          crossing?accessGranted;
                                     mproc!execute(msgRecv);
  les                              les.
```
(a) task model   *(b) task mapped model*

Figure 10. Application model

Similar to the example presented in fig. 10, any activity performed by a real-time task can be specified in POOSL and it will be followed by a similar mapping message for analysis purposes.

3.3 Platform model

The platform on which the software runs can be described as a collection of resources to perform the computations and the communication required by the application. However, as the current technique allows generation of code only for single processors architectures, only single computational resources (CPUs) were taken into account

```
SCHEDULE()() | req, oldreq : Request |
  sel
    task?execute(req);
    Scheduler scheduleRequest(req);
    SCHEDULE()()
  or
    cpu?finish(oldreq);
    if (epsilon < currentTime - oldreq getReleaseTime())
    then epsilon := currentTime - oldreq getReleaseTime()
    fi;
    req := Scheduler removeRequest(oldreq);
    if (req != nil) then cpu!execute(req) fi;
    SCHEDULE()()
  les.
```

Figure 11. Scheduler model

for the platform model in the Y-chart. As all the functionality of the application, specified in terms of tasks, must run on the same CPU, a scheduler is needed to arbitrate the access to it.

The scheduler is modelled as a POOSL process, as in fig. 11, that receives scheduling requests (data objects of type *Request*) from the newly activated tasks (the **sel** branch), or notifications from the CPU about completed requests (the **or** branch). The newly activated request is put in the list of scheduled requests by calling the data method *scheduleRequest(req)*. In the **or** branch, the scheduler receives completed requests from the CPU and checks for each of them what is the time-deviation of the implementation, by computing the difference between finishing and release time. The scheduler also removes the completed requests from the ready list by calling *removeRequest(oldreq)*, which also returns the next scheduled request, if there is one.

The data object *Scheduler* is an instance of a data class implementing the scheduling policy. Because the semantics of the model enforces atomic execution of actions, the scheduling policy is non-preemptive. Moreover, because at each virtual time moment, only actions eligible at that moment are requested by the application model and all possible orderings of the actions are allowed, a First-Come-First-Serve algorithm can be chosen. The *Scheduler* object buffers the already received requests (the activated tasks), without taking any model time, and keeps them ready for when the resource will be available for their execution. The methods of this class require as parameter a data object of type *Request*. Such a data object is built at run-time, during model execution, in order to estimate the time-deviation caused by the execution of each action in physical time on a processor.

```
RESOURCERUN()() | req: Request |
  sch?execute(req);
  delay initialLatency;
  delay req getExecutionTime()
  sch!finish(req);
  RESOURCERUN()().
```

Figure 12. Resource model

Fig. 12 presents the computational resource (CPU) model as a POOSL process receiving execution requests from the scheduler. Before the actual execution, the resource has an initial latency, which represents the overhead of the scheduling and the task context switch time. When the execution is finished, the request is sent back to the scheduler which will compute the time-deviation for the current request.

For building a completely deterministic model, the initial latencies and the execution times of requests are considered to be fixed values, equal to the worst-case situation for which there exists approaches of how to compute it for a particular CPU ([8]). In the realistic behaviour of the system, these values vary between the best-case and the worst-case. Therefore, they can be modelled using distributions, meaning that their values are probabilistically

selected at execution.

3.4 Mapping model

To analyse the performance of the system, a mapping of the application model onto the platform model is composed (fig. 13). In this approach, an explicit mapping has been chosen, represented by a POOSL communication channel linking one or more tasks to a platform, made of a CPU and a scheduler. A task is able to send execution requests to the platform, in the form of messages through the mapping channel. Such a message is encapsulated in a data object of type *Request*, together with the name of the task, its release time, and its execution time. This object arrives at the scheduler, where it is first scheduled and then sent to execution. As soon as the request is sent, the task continues its behaviour without being influenced by the execution on the processor. As according to the language semantics the communication of a message takes no model time, the timing behaviour of the application is not affected by the mapping.
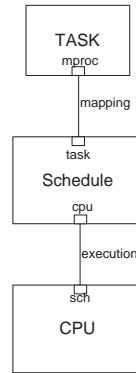

Figure 13. Mapping model

3.5 Environment model

Research in the area of timing analysis usually assumed that all external events arrive either perfectly periodic or aperiodic, without any latencies or sporadic effects. Therefore, only models of the application and the platform were typically considered for reasoning about the properties of the system.

```
ENVIRONMENT()()
  Sensor := new(Distribution) ofType(Uniform);
  while (true) do
    delay Sensor sample();
    out!event
  od.
```

Figure 14. Environment model

However, to reason accurately about the properties of an embedded system, its whole behaviour should be modelled realistically, including a probabilistic model of the environment that triggers the events. For this purpose, a discrete-event approximation of the continuous-time behaviour of the physical components can be modelled in terms of event streams occurring according to some distribution. An example of such a model is given in fig. 14. A *Sensor* is modelled in the environment to trigger a software task. The sensor is modelled as a data object of a uniform distribution type, which means that at time intervals whose length are computed based on this distribution, a new event (`out!event`) is sent to the control task.

11

3.6 Model analysis

By composing together the application, the platform and the mapping models, a complete model of a system can be built and validated using SHESim. To do design space exploration, the analysis of different application-platform combinations must be realised. This might mean choices of different processors or different decompositions of the functionality into tasks. For each combination, during the execution of the model, the scheduler can report the time-deviation that is caused by the latency of execution of actions on processors. Furthermore, based on the POOSL semantics derived from CCS, it can be detected if there is a deadlock in the system. If the time-deviation is considerable small and there is no deadlock, then the corresponding platform is a good candidate that meets the system requirements.

As shown in the previous section, the task models are designed relative to model time, not to the physical time. This approach differs from traditional approaches as the performance of the architecture or the drifts of a processor clock do not influence the timeliness of the control of the physical components in the environment anymore. As the environment "runs" relative to the virtual time, the designer is able to check if, under different circumstances, the behaviour still meets the critical deadlines.

To correctly dimension a system (the required CPU performance) such that it works in any situation, the worst case behaviour of the system must be analysed. This usually means to consider the worst-case execution times for all the activities in the system. On the other hand, the analysis of the average behaviour, based on probabilities, is also important, as it gives a measure of the suitability of the design. If the dimension of the system, needed for the worst-case situation that appears only once in a while, is far bigger than the one needed in average, that could give useful hints for a re-design.

4 A Hard Real-Time System Case Study: A Train Crossing System

A train crossing system (whose picture is presented in fig. 15) has been chosen as an example of a hard real-time control application, on which we emphasise our approach to estimate the time-deviation of the realisation from the model and how it can be dealt with. A scenario of the operation of this system is presented in [5] and more details about the system design can be found there. In this paper we will focus mainly on presenting the relevant aspects regarding the model of the system realisation.
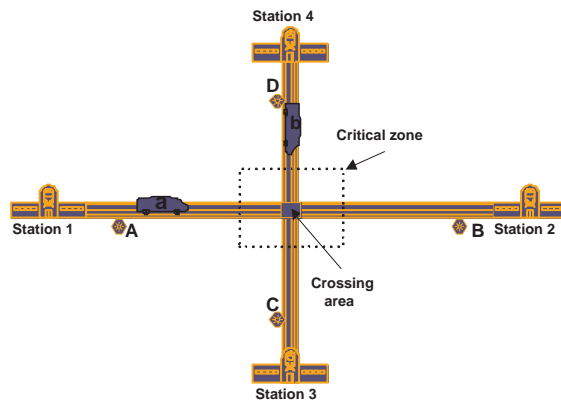

Figure 15. The train crossing system

The system consists of four train stations connected by perpendicular tracks on which two trains are running. The purpose of modelling this system is to analyse how we can built it in such way that the trains never collide and each train spends as little time as possible on its way from one station to the other. The crossing area acts as a

shared resource between the trains and there are no sensors placed in this region to detect if a train is coming. Each of the four stations has a sensor in its immediate neighborhood to detect the presence of the train. The moment when a train arrives near the crossing area is computed based on the train speed and the distance that it should have travelled from the last sensor it passed by. To compute this distance the speeds of trains are assumed constant.

In the model, the designer establishes the size of a critical zone, which represents the safety distance from the crossing from which a train has to check whether it can pass or not. If the crossing is not free, the train must stop and wait until the other one has left the crossing area. The bigger the size of the critical zone is, the longer a train has to wait for the other train to pass. The critical zone should be as small as possible to allow a train to stop *just* before the crossing area, letting the other train pass and still avoiding the trains collision. If the time-deviation between implementation and model is large, it might be the case that a collision cannot be avoided. A train may stop *within* the crossing area due to the fact that the processor cannot make sure that the checking of the crossing is done fast enough. A processor of better performance might execute all the actions required at a moment t in less time, so the error will be less and the collision can be avoided because the train stops in time.
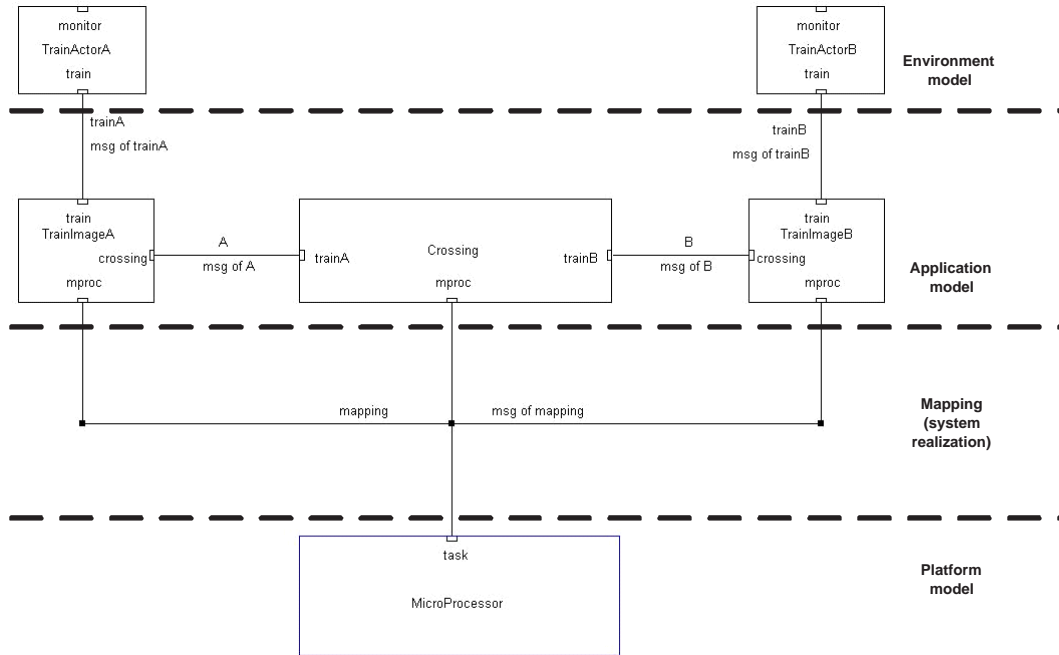

Figure 16. The model of the system realisation

In the system model, the crossing and the trains are represented as parallel POOSL processes (fig. 16). For each train in the system there is a train controller (*TrainImageA* and *TrainImageB*) that communicates with the crossing (*Crossing*) to ask for access permission. A train controller also exchanges messages with its corresponding train actor (*TrainActorA* or *TrainActorB*) to order it to stop or to get information about its position. A train actor is a discrete-event model of a continuous-time component, thus it represents a part of the controlled environment model. Only the train controllers and the crossing represent the application model and these tasks will be mapped on the target platform.

The target platform model receives notification messages from the three software tasks in the application model (the two train controllers and the crossing) about what actions they have done and that would be performed by a processor in the real system. As the system only has a small number of software tasks, it is quite obvious to be implemented on a single-processor platform.

For the target platform, we have modelled a processor that needs $1\mu s$ for each action required from a software

task in our system. By analysing the model of the realisation, we have obtained that the time-deviation between the software model and its implementation on this processor would be $6\mu s$. The delays due to the communication with the environment are already taken care of in the environmental model consisting of the *TrainActors* (the upper bound of the *CommunicationDelay* is $0.02s$). To compute the size of the critical zone we also need some numerical details about the system. The length of the crossing area is $0.044m$. The speed of the trains is assumed to be constant between two sensors if there is no stop. Train a runs with $0.470m/s$ and train b with $0.250m/s$. They need $0.045m$, respectively $0.015m$ for deceleration before standing still.

If the error between model and implementation would be zero, then the size of the critical zone could be:

$$CriticalZone = CrossingLength/2 + TrainADecelerationLength+$$
$$+TrainASpeed * CommunicationDelay = 0.06794m.$$

Due to platform given value of ϵ, the size of the critical zone must be at least:

$$CriticalZone = CrossingLength/2 + TrainADecelerationLength+$$
$$+(CommunicationDelay + \epsilon) * TrainASpeed = 0.06794282m.$$

We have computed the size of the critical zone considering the speed of train A because it is faster than train B. As it can be seen from the result obtained, ϵ is very small and practically has no influence. This means that the model can also be implemented and run on a platform of lower performance. In this case ϵ can be larger and consequently refinements must be made to the system model so that it can handle the time-deviation and make sure that the trains do not collide [5].

For safety critical systems, such as this simple example, it is crucial to eliminate all the possibilities of system's failure. The estimation of system realisation deviation from the model saves the designer from the trouble of building the actual system and exhaustively testing it on different platforms to check its functional and timing correctness.

5 Conclusions

In this paper, we have proposed an approach for estimating the time-deviation that appears in model-driven real-time software development. The approach is based on SHE method for real-time systems design and the Y-chart scheme concepts.

It is important to know the time-deviation (ϵ) that exists between a model and its implementation in order to reason about to what extent the properties verified in the model can be preserved in the implementation. This time-deviation can never be zero, but it can be decreased using a processor of better performance or compensated by a suitable system design that can bear with the error. If a suitable platform is not found, the model of the real-time application can be modified such that it takes into account the time-deviation and the resulting implementation will behave properly from safety critical reasons point of view.

The benefit of the approach presented in this paper consists in the fact that the designer is able to predict from the model what properties can be preserved and to what extent. He can also reason about possible refinements of the model to compensate the time-deviation. The approach is suitable for control-dominated applications, in which actions typically take a short amount of time, so the time-deviation usually obtained is rather small. In data-dominated applications, computations take a considerable amount of time, therefore the value of ϵ is larger. Hence future research is required to make the approach applicable to this kind of applications as well.

Another benefit of this approach is that of saving the trouble of iterating a couple of times through the generation of the real-time system's implementation and measuring the feasibility of each processor architecture. It also provides the means for design space exploration by easily modifying the architecture model. A processor architecture is described in terms of values for different actions' execution times. In order to model a different

14

processor architecture, it suffices to replace the old values with those of the new processor, so the design itself is neither removed nor replaced: the designer needs only to place the right values for the execution times of actions.

   A major future challenge of this research is triggered by complex real-time systems that require heterogeneous distributed architectures to run on. Distribution of processes in the design over multiple processors, while still satisfying the hard timing constraints of the system, is still an unsolved issue.

References

[1] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43 (1): pp. 116–146, 1996.

[2] Marc G.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Eindhoven University of Technology, 2002.

[3] Jinfeng Huang, Jeroen P.M. Voeten, Oana Florescu, Piet van der Putten, and Henk Corporaal. Predictability in real-time system development. In: *Advances in Design and Specification Languages for SoCs*. Kluwer Academic Publishers, 2005.

[4] Jinfeng Huang, Jeroen P.M. Voeten, and Marc C.W. Geilen. Real-time property preservation in approximations of timed systems. In: *Proc. of MEMOCODE*, 2003.

[5] Jinfeng Huang, Jeroen P.M. Voeten, and Piet H.A. van der Putten. Predictability in real-time system development: (2) a case study. 2004. Proc. of FDL.

[6] Jinfeng Huang, Jeroen P.M. Voeten, Andre Ventevogel, and Leo J. van Bokhoven. Platform-independent design for embedded real-time systems. In: *Proc. of FDL*, 2003.

[7] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In: *Proc. of the IEEE ASAP*, 1997.

[8] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[9] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[10] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In: *Proc. of REX Workshop*. Springer-Verlag, 1992.

[11] OMG. *Model Driven Architecture (MDA)*. OMG document ormsc/2001-07-01, Needham MA, 2001.

[12] OMG. *Unified Modeling Language (UML) - Version 1.5*. OMG document formal/2003-03-01, Needham MA, 2003.

[13] OMG. *UML Profile for Schedulability, Performance, and Time Specification - Version 1.1*. OMG document formal/2005-01-02, Needham MA, 2005.

[14] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York NY, 1994.

[15] Telelogic. TAU generation 2. `http://www.telelogic.com/products/tau/tg2.cfm`.

[16] Leo J. van Bokhoven. *Constructive Tool Design for Formal Languages: From Semantics to Executing Models*. PhD thesis, Eindhoven University of Technology, 2002.

[17] Piet H.A. van der Putten and Jeroen P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, 1997.

[18] Frank N. van Wijk, Jeroen P.M. Voeten, and A.J.W.M. ten Berg. An abstract modelling approach towards system-level design-space exploration. In: *System Specification & Design Languages*. Kluwer Academic Publishers, 2003.

16