Property-preserving synthesis for unified conrol- and data-oriented models.

Chapter 14

Property-Preservation Synthesis for Unified Control- and Data-Oriented Models

Oana Florescu, Jeroen Voeten, and Henk Corporaal
Eindhoven University of Technology and the Embedded Systems Institute
P.O. Box 513
5600 MB Eindhoven
The Netherlands

Abstract In the software/hardware engineering model-driven design methodology, preservation of real-time system properties can be guaranteed in the model synthesis up to a small time-deviation. Therefore, this methodology is well suited for the design and implementation of control systems in which execution times of actions are small; thus the time-deviations obtained are small. However, in systems containing time-intensive computations, the time-deviations become large and, consequently, the real-time properties are much weakened. This chapter proposes an approach for obtaining stronger preservation of the observable properties of the system by abstracting from its internal unobservable actions. In this way, a unified way of analysis and synthesis of a larger area of real-time applications can be obtained, which enables designers to achieve predictability in the design of many systems.

Keywords: real-time; property-preservation synthesis; observation equivalence.

1. Introduction

The main purpose of modelling is to help engineers understand the relevant aspects of a system, while avoiding the expense and trouble of actually building it. Whereas traditional forms of engineering have a well-established modelling methodology, software engineering, and particularly real-time embedded software is still an emerging discipline. Although it is applied to increasingly complex systems, its modelling techniques are neither mature nor reliable yet (Selic and Motus, 2003). Nevertheless, software models have a unique and remarkable advantage over most other engineering models; they can be used to automatically generate the realisation of the system modelled, which is an executable program for a particular platform. Starting with a simplified and highly abstract model, refinements can be carried out until a complete specification is obtained, including all the details necessary in the final product. From such a detailed specification, adequate computer tools can generate an implementation.

The model-driven architecture (MDA) initiative of the Object Management Group (Miller et al., 2001) shows that the interest in technologies for supporting model-driven development has increased. In the development trajectory proposed in MDA, system models are made from early stages to help designers in reasoning about different trade-offs. By making design decisions and adding the corresponding details to the model, the design space is narrowed. The software models are kept independent from the platform as long as possible in this design trajectory. This platform-independence provides the flexibility of reusing the design model and/or of targeting it to a different platform. Moreover, it may allow the prediction from the model of a suitable platform. Going lower in the design pyramid by increasing the number of details, a complete specification can be obtained from which the software implementation can be automatically generated.

The software components employed in the embedded systems, like the ones in cars, airplanes, printer/copier machines, or medical devices, are supposed to synchronise and coordinate different processes and activities. Therefore, their behaviour must meet hard timing constraints, either for people's safety or simply to ensure things work correctly. Usually, a real-time software component must work together with other software and hardware components to obtain the specified behaviour. Its correctness depends on both the logical result and the moment in time when the result was ready. Experience showed that existing model-driven development approaches for software systems are not suited to cope with real-time system design. Traditional design approaches proved themselves unable to capture adequately both the functional and non-functional (timing) characteristics of a system, while abstracting from low-level details. For predictably designing such systems, an appropriate methodology needs to provide (Huang et al., 2003b) (i) a suitable modelling technique that can appropriately capture functional and timing properties in models in order to formally analyse them, and (ii) a mechanism to generate the implementation from the model while preserving the properties analysed, phase also known as model synthesis.

The Software/Hardware Engineering (van der Putten and Voeten, 1997) is a model-driven design methodology suitable for analysis and synthesis of real-time systems in which actions need small execution time. In this chapter, we propose an idea for synthesis, using the same methodology, of system models containing both short actions and time-intensive computations while still preserving the real-time properties analysed. We make observations regarding the possibility of code generation from models which are equivalent from the perspective of an external user. By applying this idea, we can have a predictable and unified trajectory from a model towards a property-preserving system realisation for a large area of real-time applications (both control-oriented and data-oriented).

The remainder of this chapter is organised as follows. Section 2 discusses related research. Section 3 presents the technique used for formal modelling of systems. Section 4 shows how the properties of control-oriented system models are preserved in their implementations. Section 5 discusses a way to synthesise models of applications that contain time-intensive computations. Conclusions are drawn in Section 6.

2. Related Research

In the context of model-driven approaches for software development, the Unified Modelling Language (UML) (OMG, 2003) has been adopted as a standard facility for constructing models of object-oriented software. UML proved to be suitable for modelling the functional aspects of a system, which can also be correctly synthesised. Moreover, extensions were defined to it to provide a standardised way of denoting non-functional (timing) aspects for real-time systems as well (OMG, 2005). Nevertheless, the application of mathematical analysis techniques remains complicated due to the difficulty of relating formal techniques to UML diagrams, whereas the synthesis of the model cannot preserve the timing properties of the system.

For modelling purposes, a number of techniques and theories were proposed, targeting a certain view over a system, e.g., correctness analysis, scheduling analysis. For example, classic scheduling theory (Buttazzo, 1997) provides techniques for the analysis of timing behaviour of a system and for the scheduling of its tasks onto the target platform such that the timing constraints are satisfied. Real-time systems are assumed to be composed of independent tasks with periodic arrival times; therefore, well-studied methods, like rate monotonic scheduling, can be applied. Nevertheless, analysis of such models often yields pessimistic results and it is not suitable for handling non-periodic tasks with non-deterministic behaviours. Moreover, the models analysed by classical scheduling analysis do not incorporate information about the functionality of tasks, which makes them unsuitable for model synthesis.

A way to relax the stringent constraints on task-arrival times is by using automata with timing constraints to model task-arrival patterns. The model can describe concurrency and synchronisation of periodic, sporadic, pre-emptive, or non-pre-emptive real-time tasks with or without precedence constraints. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable (all associated tasks can be computed within deadlines). Based on the results obtained for schedulability analysis on timed automata, the TIMES tool (Amnell et al., 2003) has been designed for schedulability analysis and synthesis of real-time systems. A model consists of (i) a set of application tasks whose executions may be required to meet different timing, precedence, and resource constraints; (ii) a network of timed automata describing the task-arrival patterns; and (iii) a pre-emptive or non-pre-emptive scheduling policy. From such a model, the TIMES tool can generate a scheduler and compute the worst-case response time for all tasks. Nevertheless, TIMES tool does not have enough expressive power to describe all kinds of data computations involved in a system. This is due to the exhaustive analysis that might lead to state space explosion problems if there are many details involved. Therefore, TIMES analysis and synthesis might not scale up to any kind of system.

3. Real-Time Systems Models

The Software/Hardware Engineering (van der Putten and Voeten, 1997) is a system-level design methodology that uses a UML profile to formulate the concepts needed for the realisation of the requested functionality of a system. The UML profile smoothes the application of the Parallel Object-Oriented Specification Language (POOSL) (van der Putten and Voeten, 1997) to develop an executable model, as shown in Figure 14.1. POOSL formalises the behaviour specified in informal UML diagrams, establishing a formal executable model. The realisation of the system can be generated from this model using the Rotalumis tool (van Bokhoven, 2002).

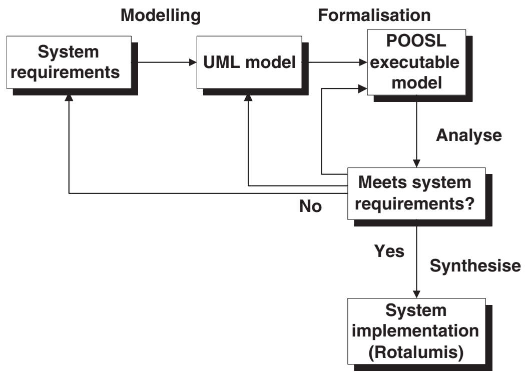

Fig. 14.1 SHE method for real-time systems design

POOSL is equipped with mathematical semantics that can formally describe concurrency, distribution, 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. The importance of the formal semantics of a modelling language in supporting the predictability of the system design process is investigated in (Huang et al., 2005).

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 Calculus of Communicating Systems (CCS) (Milner, 1989), 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, as the example in Figure 14.2 shows, where S_1–S_8 represent states of the system, a_1–a_4 action transitions, and t_1–t_3 time transitions. A timed labelled transition system represents an abstract view over a system, considering it 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. When action transitions take place, the state of the system changes by changing its content (e.g., when an event happens, certain parameters of the system get different values). In case of time transitions, only the time parameter changes its value according to the time interval specified, whereas the rest of the system content stays the same.

In a model based on the timed labelled transition system, the execution has two phases, as shown in Figure 14.3: (i) 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 (ii) by letting time pass synchronously without any action being performed (phase 2).

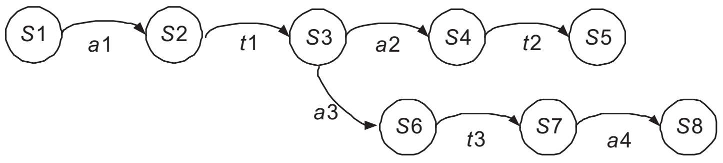

Fig. 14.2 Example of a timed labelled transition system

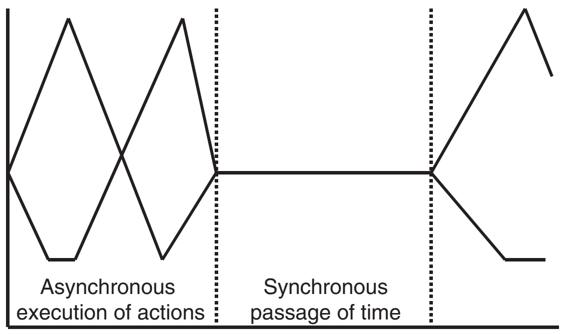

Fig. 14.3 Two phases of model execution

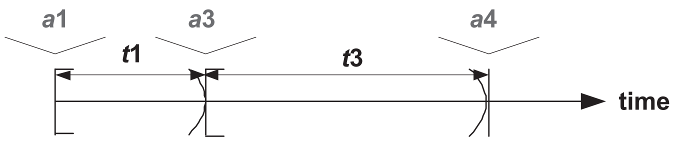

Fig. 14.4 A timed trace of the transition system

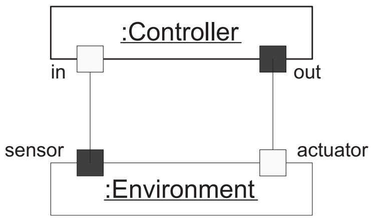

Fig. 14.5 The UML model of a simple controller

A run over a transition system represents a timed trace, as the one in Figure 14.4, where each action is executed at a particular time. As there are many possible runs due to the parallelism and non-deterministic choices that can be expressed, a POOSL model represents, in fact, a set of timed traces. If all the traces of the model satisfy a real-time property (e.g., that a particular event happens at a certain moment), then the model of a system has that particular real-time property.

For illustration purposes, a simple controller is used in the following. The UML graphical representation of this system is provided in Figure 14.5 using the UML stereotype “`capsule`”. The small black squares in the figure represent output ports and the white ones represent input ports.

Listing 14.1 POOSL model of the simple controller

```
in?input(x);         /* x is received as a message */
computation(x)(y);   /* x is the input, y is the output of ↵
    computation */
delay deadline;      /* wait for deadline units of time */
out!output(y);       /* y is sent as a message */
```

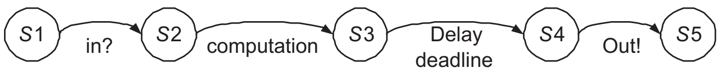

Fig. 14.6 The timed labelled transition system of the model

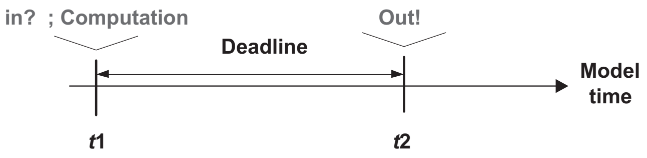

Fig. 14.7 A timed trace of the controller

The POOSL specification[1] of the system is given in Listing 14.1. The controller reads some data `x` from the environment, performs computations with it and delivers the result `y` back to the environment at a certain time.

The timed labelled transition system underlying the POOSL model looks like in Figure 14.6. According to the semantics of the language, a timed trace of the model is the one shown in Figure 14.7. `in?input(x)` and `computation(x)(y)` are executed in this exact ordering, without consuming any time and at the same instant t_1. Then, time passes for `deadline` units ($t_2 = t_1 + deadline$) and, finally, `out!output(y)` is instantly performed at t_2.

4. From a Model to Its Realisation

As mentioned in the previous section, a real-time system can be formalised as a set of timed traces. If two timed traces have the same sequence of actions, a notion of distance between them is defined. The distance represents the largest deviation between the ending points of corresponding time intervals, as shown in Figure 14.8. Two timed traces whose distance between them is equal to ϵ are called ϵ-close. If two execution traces are ϵ-close and one of the traces satisfies

[1]Note that the notations in a POOSL specification are CCS alike.

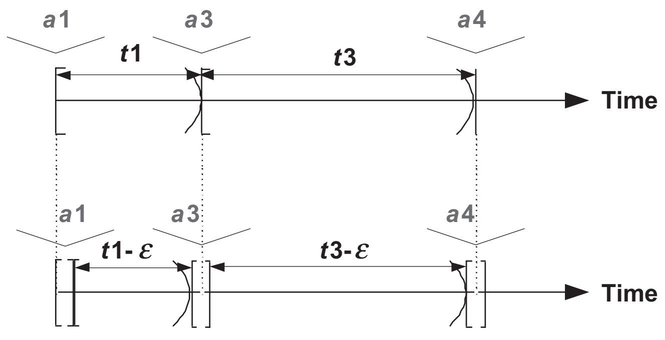

Fig. 14.8 Timed traces ϵ-close

a real-time property,[2] then this property, weakened up to ϵ,[3] is satisfied in the second trace as well. This result was mathematically proved in (Huang et al., 2003a).

Both the model and the realisation of a system can be viewed as sets of timed traces. To obtain an implementation of a system that preserves the properties analysed in its model, thus an implementation consistent with the model, two things must be achieved: (i) to generate a trace in the implementation from the set of execution traces of the model, and (ii) to make the corresponding traces in the model and in the implementation to be ϵ-close.

A mechanism of generating a trace from a POOSL model was proposed and proved correct in (Geilen, 2002). The data part of a POOSL model is directly translated into corresponding C++ expressions. Each process in the model is represented by a C++ data structure named process execution tree (PET), whose nodes represent statements in the specification of behaviour. During the evolution of the system, a PET scheduler makes choices for granting actions or time transitions, while each PET adjusts its internal state according to the choice of the PET scheduler. This mechanism guarantees that the realisation of the model generated by the code generation tool is a trace from the model.

Although actions in a model are regarded timeless, in reality, it will always take a certain amount of time to execute them; between the corresponding traces there appears a *time-deviation*. If the distance between these two traces is ϵ (ϵ-hypothesis), then *all* the properties of the model are preserved up to ϵ in the implementation.

To generate the implementation ϵ-close to the POOSL model, the code generation tool, Rotalumis (van Bokhoven, 2002), synchronises the *model time*

[2]An example of a real-time property is that a certain action happens at a particular moment in time.

[3]If a property P is true in the first trace in the interval $[t_1, t_2]$, the other trace satisfies P in the interval $[t_1 - \epsilon/2, t_2 + \epsilon/2]$.

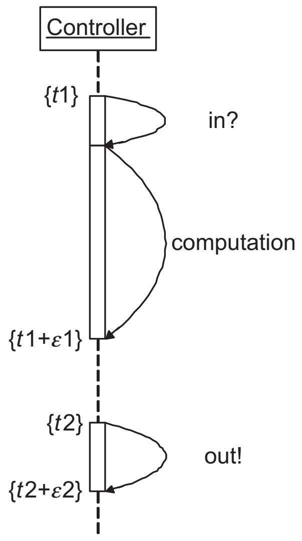

Fig. 14.9 Implementation of the controller in physical time

with the *physical time*. As shown in the UML sequence diagram from Figure 14.9, all the actions that happen instantly in the model at a certain time t (in Figure 14.7, `in?` and `computation` happen at model time t_1, `out!` at t_2) are executed within a small ϵ amount of time around the corresponding moment in physical time (`in?` and `computation` are executed in ϵ_1 around physical time t_1, `out!` in ϵ_2 around t_2). To maintain the synchronisation between model time and physical time, **`delay`**s are not executed in the implementation exactly as specified in the model (*deadline* units of time), but physical time passes until the next corresponding moment in the model time is reached (the delay *deadline* $= t_2 - t_1$ is shortened to $t_2 - t_1 - \epsilon_1$).

The size of the maximum time-deviation between a model and its implementation can be obtained at the time of generation and execution of code by using measurements. On the other hand, it can be estimated from the model itself, using the Y-chart scheme (depicted in Figure 14.10) approach for design space exploration. This scheme contains the models of both the real-time application and the target platform, and by analysing their mapping, as shown in (Florescu et al., 2004b), the time-deviation can be monitored. This deviation depends on how many actions need to be executed at the same time in the model as well as on their execution times. If the value obtained for ϵ is considered too large, either the implementation is generated for a higher performance platform on which the execution of all the actions takes less time, the mapping is changed, or the model is redesigned.

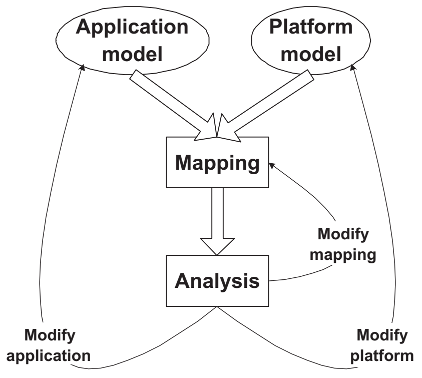

Fig. 14.10 Y-chart scheme for real-time systems design

5. Realisation of Systems with Time-Intensive Computations

In a model of a real-time system, usually a distinction can be made between *actions* and *time-intensive computations*. *Action* is the name given to an "activity" specified in the model that needs small execution time on the target platform (e.g., a control action). On the other hand, *time-intensive computations* are the "activities" specified in a model that usually need a considerable amount of time for execution, as it is the case of the `computation` in Figure 14.9. In case of real-time systems containing such computations, the time-deviation between the model and the implementation is usually large. Therefore, with the current generation of code, the properties analysed in the model will be much weakened in the implementation.

Nevertheless, in data-oriented real-time applications, many computations that take considerable amount of time are modelled (for example, different multimedia algorithms must be applied on a stream of data). For this kind of system, it is not intended for the computations to be instantaneous, but to be finished before a *deadline* when the results must be given to the environment (like in the example given in Section 3).

Two systems are called observational-equivalent if they cannot be distinguished between them through the interaction of a user with each of them. They have the same observable properties[4] and the same set of timed traces with respect to these properties. Therefore, an implementation preserving the observable properties of a model preserves the observable properties of the observational-equivalent one.

[4]A user can see the same properties by interacting with the systems.

Listing 14.2 Observational-equivalent model of the controller

```
in?input(x);
computation1(x)(y1);
delay deadline1;
computation2(y1)(y2);
delay deadline2;
computation3(y2)(y);
delay deadline3;
out!output(y);
```

Based on this insight, in case of systems with time-intensive computations, instead of generating an implementation for the original model we could generate the implementation for an observational-equivalent one. In Listing 14.2, we give a specification, which is observational-equivalent with the example given in Section 3. The `computation` is split, for example, into three smaller parts (`computation1`, `computation2`, and `computation3`) that, put in sequence, form the original computation specified in the model. After each small computation, a certain amount of time delay follows (`deadline1`, `deadline2`, and `deadline3`) and the sum of all delays makes the original delay amount ($deadline = deadline_1 + deadline_2 + deadline_3$). A timed trace of this system is given in Figure 14.11.

The two systems modelled are, obviously, observational-equivalent for a user for whom it is important when the input data `x` is read from the environment, what is the flow of computations performed on `x`, and when the final result `y` is available. For this system, the existing synthesis mechanism for POOSL models, which relies on the ϵ-closeness between traces for the properties-preservation, can be applied. To obtain an implementation trace ϵ-close to its corresponding trace in the model, as shown in the previous section, a synchronisation of each moment in the model time when an action happens with the corresponding physical time, up to ϵ, is realised, as shown in Figure 14.12. For the implementation of the original model, there are only two synchronisation points, t_1 and t_2 from Figure 14.9, and the time-deviation is

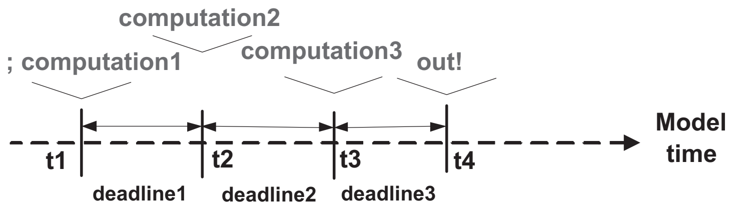

Fig. 14.11 Observational-equivalent model timed trace

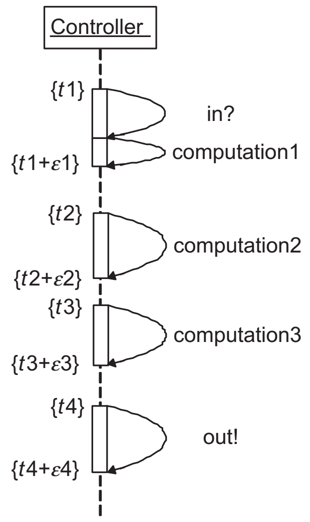

Fig. 14.12 Implementation of the equivalent model in physical time

large. For the observational-equivalent model, in Figure 14.12, there are four synchronisation points, t_1, t_2, t_3, and t_4, and the time-deviation for each of them is smaller. Therefore, over the whole system, the properties are stronger preserved in the realisation of the second model.

From the perspective of the code generation tool, looking at Figure 14.12, what it actually has to do is to generate a trace in which the execution of the `computation` (made of `computation1`, `computation2`, and `computation3`) starts immediately after reading `x` from the environment, continues more or less without stopping, and finishes before the moment the result `y` must be given back to the environment. In other words, `deadline` represents the *deadline* of the computation, and only the observable actions of the system are synchronised in the physical time as depicted in Figure 14.13. The time needed for the execution of computation does not have to count against the size of the time-deviation between model and implementation because, for the observational-equivalent model in Figure 14.12, the value of ϵ is small and it is determined by the execution time of `out!output(y)`.

As shown in this simple example, the implementation of a model containing time-intensive computations can be generated from an equivalent model that has the same observational behaviour and the same properties as the original one. Under these circumstances, we can define *actions* and *computations* slightly different than at the beginning of this section. We name *actions* those activities that can be observed by a user interacting with the system and,

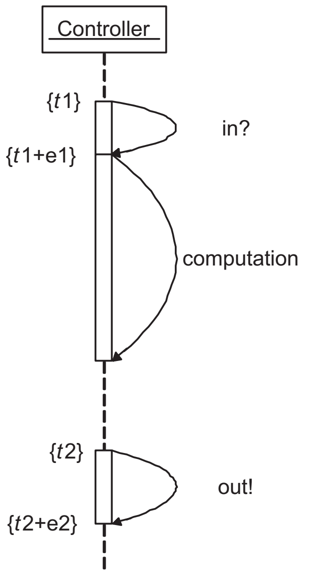

Fig. 14.13 A possible execution that still preserves the properties

therefore, their moments of execution in the model time must be synchronised with the physical time. On the other hand, *computations* are the internal activities of a system that a user cannot observe and who need to be scheduled for execution such that they can meet their deadlines. Moreover, if they are still running, they can be pre-empted by an action whose model time must be synchronised with the physical time. By abstracting from the internal actions of the model and synchronising the model time with the physical time only for the moments when observable actions happen, the observable properties of a model can be preserved; thus the code generation tool can handle the model synthesis of data-oriented applications as well.

However, it is not always possible to execute a computation within a *deadline*. For a specification like the one given in Listing 14.3, according to the formal semantics of the language, the urgent message can arrive either before the execution of `computation(x)(y)` or during the **`delay`**, which means after the execution of `computation` has finished. If the `computation` has a `deadline` in the implementation, then, at the time the `urgentMessage` appears, the execution of `computation` must be pre-empted. The `computation` will not be allowed to continue; thus the state in which the system realisation will be in that moment will not be a state present in the model. In this case, the relaxation of the timing constraints is not possible because there is no equivalence relation between this model and another one that has the computation split into smaller parts. Therefore, the execution time of the computation

Listing 14.3 Example of model without observational equivalence

```
abort
  (computation(x)(y); delay deadline)
with p?urgentMessage;
```

contributes to the total time-deviation between the model and the implementation of this system.

Nevertheless, the mechanism that we propose in this chapter for the synthesis of real-time systems with time-intensive computations has the benefit of using an existing methodology, without changing the syntax, the semantics of the modelling language, or anything else, just by relaxing the constraints on the properties to be preserved. However, work needs to be done to formalise these ideas and to mathematically prove them. Moreover, a mechanism of identification of the observational-equivalent system whose implementation is the same with the one of the given model is required.

To analyse a model with different kinds of activities (taking longer or shorter execution time), the Y-chart scheme can be used again. Such a unified model helps designers in reasoning about aspects like what is the largest time-deviation (ϵ) that the system can allow, or what is an appropriate scheduling of the time-intensive computations, as shown in (Florescu et al., 2004a).

6. Conclusions and Future Work

To achieve a predictable design of real-time embedded systems, a unified executable model, capturing both functional and timing aspects of a system, is suitable to allow engineers to reason about different properties in a unified manner. Moreover, such a model must be easily refinable towards a complete system specification from which the implementation can be automatically obtained.

In this chapter, we have presented how the Software/Hardware Engineering methodology can be used for the modelling, analysis, and synthesis of a large area of real-time systems (control-oriented, data-oriented applications). The POOSL modelling language allows specification of both timing and functional aspects of systems, whereas the ϵ-hypothesis guarantees the preservation of properties between two timed systems with a small time-deviation. By satisfying the ϵ-hypothesis, the code generation tool, Rotalumis, succeeds in synthesising an implementation of a model preserving *all* the properties, in case the actions specified are not time-consuming. For the data-oriented applications, we propose a way to generate the realisation from a model that is observational- equivalent with the original one but which has the advantage that the time-deviation obtained for it is smaller. In fact, we suggest that it is possible to make an abstraction from the internal actions of the system and synchronise the physical time with the model time only for the observable actions. Moreover, this realisation would preserve the observable properties of the original real-time system.

For the future research, we aim at formalising this mechanism and at giving a mathematical definition for the circumstances when computations can be safely pre-empted by actions. Furthermore, we want to adapt the code generation tool to work according to the proposed mechanism and to apply it to realistic case studies.

Acknowledgments

This work is being carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

References

Amnell, Tobias, Fersman, Elena, Mokrushin, Leonid, Pettersson, Paul, and Yi, Wang (2003) TIMES: a tool for schedulability analysis and code generation of real-time systems. In: *1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS) 2003.*

Buttazzo, Giorgio (1997) *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Boston, MA.

Florescu, Oana, Voeten, Jeroen, and Corporaal, Henk (2004a) A unified model for analysis of real-time properties. In: *1st International Symposium on Leveraging Applications of Formal Methods (ISoLa) 2004.*

Florescu, Oana, Voeten, Jeroen, Huang, Jinfeng, and Corporaal, Henk (2004b) Error estimation in model-driven development for real-time software. In: *Forum on Specification & Design Languages (FDL) 2004.*

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

Huang, Jinfeng, Voeten, Jeroen, and Geilen, Marc (2003a) Real-time property preservation in approximations of timed systems. In: *1st ACM & IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2003).*

Huang, Jinfeng, Voeten, Jeroen, Ventevogel, Andre, and van Bokhoven, Leo (2003b) Platform-independent design for embedded real-time systems. In: *Forum on Specification & Design Languages (FDL) 2003*.

Huang, Jinfeng, Voeten, Jeroen, Florescu, Oana, van der Putten, Piet, and Corporaal, Henk (2005) *Advances in Design and Specification Languages for SoCs (Best of FDL'04)*, chapter Predictability in real-time system development. Kluwer Academic Publishers, Boston, MA.

Miller, Joaquin, Mukerji, Jishnu, et al. (2001) Model driven architecture (MDA. Technical report, OMG document ormsc/2001-07-01, Object Management Group (OMG), Needham, MA.

Milner, Robin (1989) *Communication and Concurrency*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

OMG (2003) *OMG Unified Modeling Language (UML) Specification—Version 1.5*. OMG document formal/2003-03-01, Object Management Group (OMG), Needham, MA.

OMG (2005) *UML Profile for Schedulability, Performance, and Time Specification—Version 1.1*. OMG document formal/2005-01-02, Object Management Group (OMG), Needham, MA.

Selic, Bran and Motus, Leo (2003) Using models in real-time software design. *IEEE Control Systems Magazine*, 23(3).

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

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