Chapter 8

PREDICTABILITY IN REAL-TIME SYSTEM DEVELOPMENT

Jinfeng Huang, Jeroen Voeten, Oana Florescu, Piet van der Putten and Henk Corporaal

Faculty of Electrical Engineering Eindhoven University of Technology
5600MB Eindhoven, The Netherlands

J.Huang@tue.nl

Abstract The large gap existing between requirements and realizations has been a pertinacious problem in complex system design. This holds in particular for real-time systems with strict timing constraints and critical-safety requirements. Designers have to rely on a multi-step design process, where design decisions are made at different modelling levels. To ensure the effectiveness of this design process, predictability should be well-supported by design approaches, allowing designers to predict properties of future design outcomes based on existing design results. In this chapter, we first discuss the role of the semantics of design languages and investigated how they can support a predictable design process. Then, the deficiencies, w.r.t. predictability support, of existing design approaches for real-time systems are illustrated by an example. Finally, a predictable design approach for real-time systems is introduced to overcome this problem.

Keywords: Real-time, predictability, semantics, compositionality, composability

Introduction

The aim of real-time system design is to fill the gap between requirements and the realization. However, due to the continuous increase of the functional complexity of real-time systems, and because of stringent timing requirements they have to satisfy, the design gap has increased tremendously. Since traditional code-centric design approaches are obviously not capable of coping with this increasing complexity, designers have to resort to a multi-step design process, where the system is specified and analyzed at different levels of abstractions (see Figure 8.1). This design process usually involves *requirement*

P. Boulet (ed.) Advances in Design and Specification Languages for SoCs, 123–139.

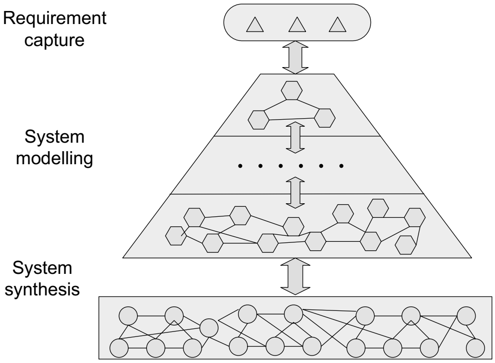


Figure 8.1. The multi-step design

capture, *system modelling*, and *system synthesis*. During *requirement capture*, the system is specified at the most abstract level, which defines the needs and constraints of the system. During *system modelling*, designers explore the design space at different abstraction levels, make design decisions through successive design steps and finally propose a proper design solution, which serves as a blueprint to synthesize a realization. During *system synthesis*, a model is transformed into a realization, which is expected to meet desired properties.

To smoothen the design process and improve productivity, consistency between design outcomes has to be maintained during each design step. In other words, predictability should be well-supported by design approaches, allowing designers to predict properties of future design outcomes based on existing design results.

The remainder of the paper is organized in four sections. In Section 1, We show that semantics of a design language plays an important role for the multi-step design process and has a direct impact on the support for predictability. In Section 2, we will briefly explain the deficiencies of existing approaches in supporting predictability during the design of real-time systems. To solve the problem presented in Section 2, we introduce a predictable design approach for real-time systems in Section 3. Section 4 concludes this chapter.

1. Semantics of design languages

Semantics of design languages has a direct impact on the thinking pattern of developers and the meaning of design outcomes. According to the different abstraction levels of design thoughts, three categories of design languages, requirement, modelling and implementation languages, are involved in the design process.

1.1 Requirement languages

Requirements express the needs and constraints that are put upon a system, each of which is a property that must be present in realizations in order to satisfy the specific needs of some real-world application [Kotonya and Sommerville, 1998]. Usually, requirements are written in natural languages. However, due to the ambiguity of natural languages, complex concepts are usually very difficult to specify precisely. This can result in errors and iterations during the design process. Formal semantics is proposed as a solution to solve the ambiguity problem. It is embedded in requirement languages to promote understandability of requirement specification, to facilitate the automatic checking of requirement consistency and completeness, and to improve the traceability of requirements during the multi-step design process [1].

1.2 Modelling languages

System modelling is the most challenging and creative activity of the design process. During system modelling, designers need first to understand thoroughly the requirements, carefully explore the design space and finally devise a design solution (model). The design model serves as the basis for later system synthesis, the success of which depends to a large extent on the model itself.

Due to the potential complexity of real-time systems, the modelling of such a system is often accomplished by taking a number of steps. Each step only considers a part of the system that is relevant to address some specific design problems. In addition to possessing adequate expressive power to assist designers to specify desired aspects of the system and to analyze the system behavior of interest at each design step, the semantics of a modelling language should also support effective model transformations, which preserves properties of interest during the multi-step design process.

Model transformations: abstraction and refinement. Abstraction and refinement are two elementary transformations performed during the design process (as shown in Figure 8.2). Abstraction is the activity that tries to remove (or hide) irrelevant information, which improves the comprehensibility of existing design models and facilitates the evaluation of different design solutions. The major goal of the abstraction activities is to improve the understandability of the design, enabling design decisions to be made. Refinement is the activity that adds more implementation details to models, thereby reducing the gap between models and realizations. The major goal of refinement activities is the implementability. Intuitively speaking, abstraction activities intend to clarify what the system (component) can do, while refinement activities intend to clarify how the functionality of the system (component) can be achieved.

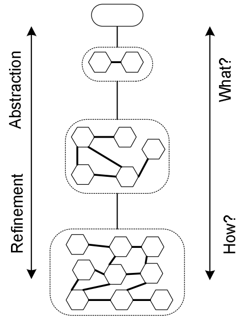


Figure 8.2. Basic design activities

A design process can be considered as a set of abstraction/refinement activities, which intends to fill the gap between the desired properties (what the system should be) and the realization (how the system functions). The effectiveness of model transformations can significantly affect the required design time and cost. This holds in particular for large-scale systems.

In practice, compositionality (or composability) is often regarded as an important characteristic that the semantics of a modelling language should possess, in order to facilitate model transformations for complex systems, where model transformations can be carried out on its subsystems.

Compositionality. The well-known principle of compositionality [Partee et al., 1990] states that *the meaning of a design description is a function of the meanings of its parts and of the syntactic rules by which they are combined.* It is originally proposed to guide the association of the semantics and the syntax of a design language and to assist designers in understanding the meaning of a complex design description in a structured way.

Consider that a system (or subsystem) is represented by a tree structure, where each leaf is a syntactic primitive and other nodes are combination rules. Compositionality ensures that each syntax sub-tree can be understand independently without the consideration of other parts of the tree. Due to the potential complexity of the syntax tree, the semantic interpretation of a complex design description can be far from simple. We can easily foresee that the interpretation of a syntax tree with hundreds of levels, which is not unusual for a complex design description, could easily grow beyond human's understanding. Therefore, compositionality alone does not promise that the meaning of a recursively composed syntax tree can be understood easily.

However, when compositionality is applied to model transformations (abstraction/refinement), it offers many benefits to reduce design complexity and to improve design efficiency. Compositional semantics divides a complex sys-

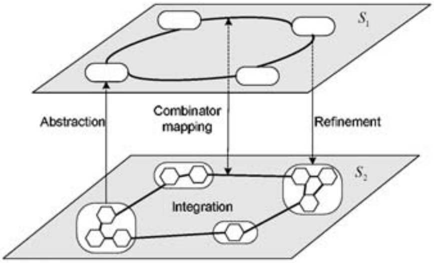


Figure 8.3. Abstraction/refinement based on the compositional semantics

tem into a set of semantic components and ensures the semantical independency of each component in the system. Thus, abstraction/refinement concerning of the whole system can be achieved by local abstraction/refinement for each component and the mapping of combinators to corresponding ones in the other abstraction/refinement level (see Figure 8.3). Furthermore, the correctness of the abstraction/refinement activities can also be verified locally.

One example of design languages equipped with compositionality semantics is CCS (Calculus of Communicating Systems) [Milner, 1989]. Based on the compositional semantics of CCS, observation equivalence is defined, which states that two models are observational equivalent if and only if both models exhibit the same communication behavior to the external observer. The semantic equivalence relation provides the theoretical basis for transformational design approaches, where components of a high-level model are iteratively refined into equivalent components with more details. In this way, observation equivalence can effectively assist the abstraction/refinement of a design description. More detailed discussion about transformational design approaches can be found in [van der Putten and Voeten, 1997], [Koomen, 1991]. Example 8.1 illustrated how abstraction/refinement activities can be carried out in CCS.

EXAMPLE 8.1 *Suppose that system S consists of two components P and Q, which are depicted by:*

$$P \equiv (a.b \parallel \bar{b})\backslash b,\ Q \equiv (c.d \parallel \bar{c})\backslash c \text{ and } S \equiv P \parallel Q. \qquad (8.1)$$

The semantics of CCS allows designers to consider the abstraction of P and Q independently from each other. In other words, no matter in what context that P or Q are embedded, P can always be abstracted as $P' \equiv a$ and Q as $Q' \equiv d$. An abstraction of S can be $S' \equiv a \parallel d$. Conversely, P, Q and S are possible refinements of P', Q' and S' respectively.

In summary, suppose $S \equiv P_1 \oplus P_2 ... \oplus P_n$ is a system expressed by a language with a compositional semantics, where P_1, P_2... P_n are components of

S and where $\oplus$ is a combinator of components. The compositional semantics guarantees that abstraction or refinement P'_1, P'_2... P'_n of P_1, P_2... P_n can be carried out independently. Therefore an abstraction or refinement of S can be $S' \equiv P'_1 \odot P'_2 ... \odot P'_n$, where $\odot$ is the corresponding mapping of combinator $\oplus$ in S. In Example 8.1, $\oplus$ and $\odot$ are both the parallel compositional combinator $\|$. In practice, S' can be expressed in the same language as that used for S or in a totally different language. For example, properties of a system written in a requirement language can be abstractions of a system written in a modelling language (see the next subsection *composability*).

Composability. The concept of compositionality is intuitively useful in achieving effective abstraction/refinement during the design of complex systems. However, in practice, it is not always as effective as expected. An important reason is that it does not put any restrictions on the assignment of meaning to combinators. As a consequence, semantical independency can always be achieved by assigning trivial semantics to combinators [Zadrozny, 1994]. In practice, the combinator semantics for both abstractions and refinements should be simple enough. For example, the semantics of the combinators $\|$ and $+$ in CCS is defined in a natural way and can be understood easily. The abstraction/refinement of sub-processes in CCS also retains the original combinators.

In the context of concurrent systems, a more restricted “version" of compositionality is sometimes called *composability*. Composability states that properties satisfied by individual components of a system should be satisfied by their parallel compositions [Sifakis, 2001]. For example, assume reactive system S consisting of two parallel components P and Q has a timing response property φ, which states that every environmental stimulus p must be followed by a response q within 3 seconds. If P satisfies φ and the design language supports composability, then $S \equiv P \parallel Q$ should also satisfy φ.

More generally, consider a system $S \equiv P_1 \parallel P_2 ... \parallel P_n$ expressed by a language supporting composability, where P_1, P_2... P_n are components of S and $\|$ is the parallel combinator. Assume each component P_i satisfies property φ_i respectively. Composability of a design language states that S satisfies the simple logical conjunction of these individual properties ($\varphi_1 \wedge \varphi_2 ... \wedge \varphi_n$). We can see that only the parallel operator ($\|$) and the logic conjunction ($\wedge$) are used in composability and their semantics are defined independently from the semantics of composed components.

1.3 Implementation languages

System synthesis is an activity that converts a model into a complete system implementation while preserving the correctness of the model. During this stage, the system is often depicted by an implementation language (such as Java, C and C++), the semantics of which is usually related with and constrained by the target platform. Due to the different notions and assumptions made at the modelling stage and at the implementation stage, it is not always straightforward to correctly transform a model into a realization. As a consequence, it is difficult to guarantee the validity of the realization w.r.t. the satisfaction of the desired properties, which have been verified in the model.

The difficulty of maintaining correctness between a model and its realization is attributed to several reasons. First, during the modelling stage, certain assumptions are often made about the semantics of modelling languages in order to effectively explore the design space. These assumptions are valid at certain abstraction levels, but they do not always hold for the semantics of implementation languages. For example, to facilitate the analysis of the timing behavior of a model, it is often assumed that actions are instantaneous. However, every action does take a certain amount of execution time in every implementation language. Without carefully considering this difference during system synthesis, the realization may exhibit an entirely different behavior than the model does. Second, some primitives and operations defined in modelling languages do not have direct correspondences in implementation languages. For example, during system synthesis, parallel operations in the model is often implemented by means of a specific thread mechanism offered by the target operating system, which semantics is not always consistent with that of the modelling language.

In most existing design approaches for real-time systems, system synthesis is achieved mainly by a syntactic mapping, instead of by a semantic mapping between the modelling and the implementation language. As a result, the synthesized realization may exhibit a different system behavior than the design model does. A more detailed investigation of real-time system synthesis will be presented in Section 2.

2. Real-time system design approaches

In this section, we are going to evaluate whether existing design approaches have adequate semantic support for real-time systems. We classify existing approaches into two categories, platform-dependent approaches and platform-independent approaches, based on the different timing concepts adopted. This is a justifiable classification because approaches adopting the same concept of timing often provide similar predictability support during the design process. Briefly speaking, platform-independent approaches use a system variable to represent time (denoted as the *virtual time*), while platform-dependent approaches adopt the *machine time* to represent time progress. This implies that the timing behavior of a system depends on the underlying computing platform.

2.1 Platform-dependent design approaches: ineffective model transformations

Platform-dependent approaches take platform computation constraints into considerations at the modelling stage, and use the machine time to specify the timing behavior in their modelling languages. Examples of these languages are Rose-Rt [RoseRT], [Selic et al., 1994] and SDL-96. One major advantage of using the machine time is that no extra (or new) timing concepts are introduced other than those adopted in imperative languages such as C and Java. These approaches are readily accepted by designers, who are familiar with imperative languages. However, this timing semantics is often too ambiguous to support model transformations. This is illustrated by the following example.

EXAMPLE 8.2 ***Two synchronized processes*** P ***and*** Q***:*** *Consider a simple real-time system (shown in Figure 8.4) consisting of two parallel processes* P *and* Q *(*$P \parallel Q$*), each of which comprises an iterative code segment involving timed actions. At the beginning of each iteration,* P *and* Q *synchronize with each other. Then process* P *sets a timer with a 3-second delay and process* Q *sets a timer with a 2.999-second delay. After the timer of* Q *expires,* Q *sends a “rpl-sig" message to* P*. For process* P*, there are two possibilities: 1)*P *receives the timer expiration message and outputs the message “wrong"; 2)*P *receives the reply message from* Q*, resets its own timer and outputs the message “correct".*

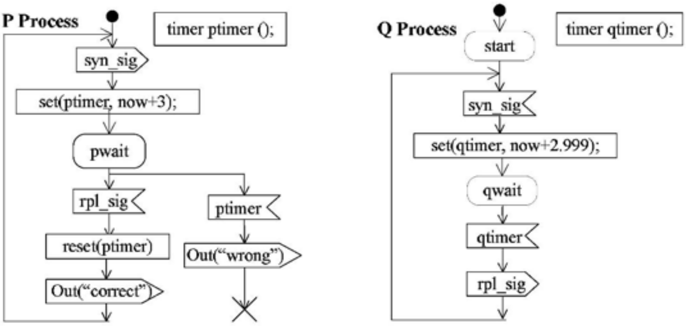
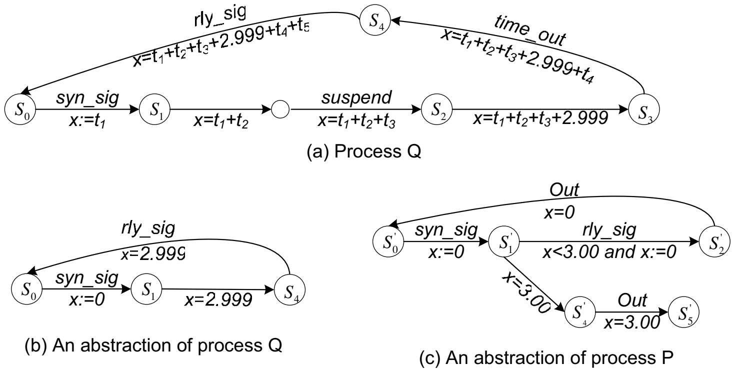


Figure 8.4. A system with two parallel processes P and Q

Here, we use a graphical modelling language based on SDL-96 to describe the system (shown in Figure 8.4). In SDL-96, the timing semantics is given in such a way that each action takes an undefined amount of physical time [2] [Graf, 2002] and the interpretation of timing expressions (such as timers) relies on an asynchronous timer mechanism provided by underlying platforms [Leue, 1996].



(a) Process Q

(b) An abstraction of process Q

(c) An abstraction of process P

Figure 8.5. The semantics of process P and Q

Suppose two processes P and Q are designed separately, which is often the case in complex system design. Now, let us first look at the timing semantics of process Q depicted in Figure 8.5(a), where x is a clock used to express timing constraints on actions and $x := 0$ represents the setting of clock x to zero[3]. The process first receives a “syn-sig” message, which takes time duration t_1. Before the next statement $(set(qtimer, now + 2.999))$ is executed, the operating system might switch to other processes taking a total amount of time t_2, before it switches back to process Q. Then the timer is set and the process is suspended (taking time t_3) to wait for the timer expiration message. Between the time that the timer expires and the time that process Q responds to the $time\_out$ message, again the operating system might take a total amount of time t_4 for the execution of other processes.

Execution times t_1, t_3 and t_5 are neglectable w.r.t. most of real-time properties of interest in a modern computing platform. In the case that process Q is the only active process running on the platform, t_2 and t_4 are zero. As a consequence, Figure 8.5(b) can be considered to be a proper abstraction of process Q. In a similar way, an abstraction for process P can be obtained, which is depicted in Figure 8.5(c). In design practice, it is often assumed that compositionality (or composability) is well supported. That is, the integration of parallel processes can preserve the properties of the integration of their abstractions. Therefore, the integrated system ($P \parallel Q$) is often reasoned about through the abstractions. This would indicate that P should never output the ”wrong" message when it is integrated with process Q.

However, in certain circumstances, the platform-dependent semantics of both processes does allow process P to output the “wrong" message in the integrated system. For example, in Figure 8.5(a), when process Q is in state S_1, the underlying operating system can first make P the active process, then P can set the timer and suspends itself, after which the operating system switches back to Q, which sets a timer with an expiration of 2.999 seconds. If one context switch, one timer setting, suspending one process together with the other necessary scheduling execution take more than 0.001 seconds in total [4], the timer of process P might expire before that of process Q. As a result, P outputs the “wrong" message.

From the above example, we can see that the abstraction of the integration of a set of components cannot always be correctly reasoned about from the abstraction of its components. To eliminate the unexpected behavior, designers have to rely on ad-hoc way to tune the behavior of each component, involving a tremendous number of design details of other components to be considered. As a result, the design process is often time consuming and prone to errors.

Real-time scheduling is often adopted in practice to alleviate the problems mentioned above for platform-dependent design approaches.

Real-time scheduling:. In the research domain of real-time scheduling, a system is viewed as a set of concurrent tasks. A scheduler is used to manage the activation and execution of tasks concurrently running in the system. It assigns the computation time by giving different priorities to tasks. In general, the task with a higher priority is scheduled before those with lower priorities. The goal of real-time scheduling is to devise a priority assignment scheme to ensure that every task can be accomplished in time. In principle, a feasible schedule can eliminate unwanted interferences from other tasks, reducing the ambiguity of the timing semantics of each task. However, real-time scheduling lacks a consistent framework to integrate functionality and timing [Liu and Joseph, 2001], and it is only suitable for a particular set of real-time systems, such as periodic systems. Hence they are not a general solution to the design of complex real-time systems. Especially interaction-intensive real-time systems are difficult to design with scheduling theory.

2.2 Platform-independent design: ineffective system synthesis

Contrary to platform-dependent design approaches, platform-independent design approaches often adopt a virtual timing concept, which is independent of any underlying execution platforms. Furthermore, the semantics of their modelling languages often treats the time progress and the action execution in an orthogonal way [Nicollin and Sifakis, 1991], which can reduce the ambiguity of the timing semantics and improve the understandability of design descriptions. In this semantic framework, system actions (such as communications and data computations) are timeless (taking zero time) and time passes without any action being performed. On one hand, such semantics can provide sufficient expressive power to describe the timing behavior of a system.

On the other hand, compositionality (or composability) is supported by the semantics of their modelling languages and effective abstraction/refinement can be supported during the design process. Furthermore, since actions are instantaneous, additional analysis code does not take up time, keeping the original timing behavior of the system unchanged. A typical modelling language based on this semantic framework is SDL-2000 [Z.100, 2000], which is supported by the TAU Generation 2 tool (TAU G2 in short) released by Telelogic [TAU G2]. Other examples often used in academic contexts are timed automata or process algebra, such as timed CCS.

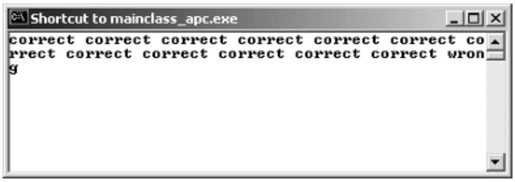
The timing semantics of the design descriptions in Example 8.2 can also be given in platform-independent semantic frameworks. In these frameworks, t_1 till t_5 are all zero and we can always consider the semantics depicted in Figure 8.5(b) for process Q to be a proper abstraction of that in Figure 8.5(a), and the same holds for the abstraction of process P depicted in Figure 8.5(c). Consequently, the abstraction of the combined system $P \parallel Q$ can be captured by combination of Figure 8.5(b) and Figure 8.5(c), in which process P should never output the “wrong" message. We made the same model in TAU G2, and the behavior of the system ($P \parallel Q$) was indeed as expected.

Although most platform-independent approaches provide sufficient support in their modelling languages for predictable design, bridging the large semantic gap between these modelling languages and implementation languages is still not solved adequately (see Section 1, *Implementation languages*).

Automatic transformation of design models to realizations is a superceding technique to manual transformation, the latter of which is inefficient and prone to errors. In current practice, the automatic transformation is achieved mainly by the syntactic mapping of syntax primitives and constructs between two design languages, instead of by a semantic mapping. As a result, inconsistencies can be observed between the design model and the realization. For example, actions are usually assumed to be instantaneous in the model, while they do take a certain amount of physical time in the realization. Without careful considerations of this semantic difference, the realization can exhibit faulty behavior. Although the model in Example 8.2 made in TAU G2 is proven to be correct, errors are observed in the automatically synthesized realization (see Figure 8.6).

3. A predictable design approach

In the previous section, we have investigated the deficiency of the existing design approaches in supporting predictability for real-time systems. In this section we introduce a design approach which can overcome this problem. This approach has two distinct characteristics. First, the POOSL language is adopted at the modelling stage, which is a platform-independent modelling



```
correct correct correct correct correct correct co
rrect correct correct correct correct correct wron
g
```

Figure 8.6. The output of a realization of two parallel processes P and Q

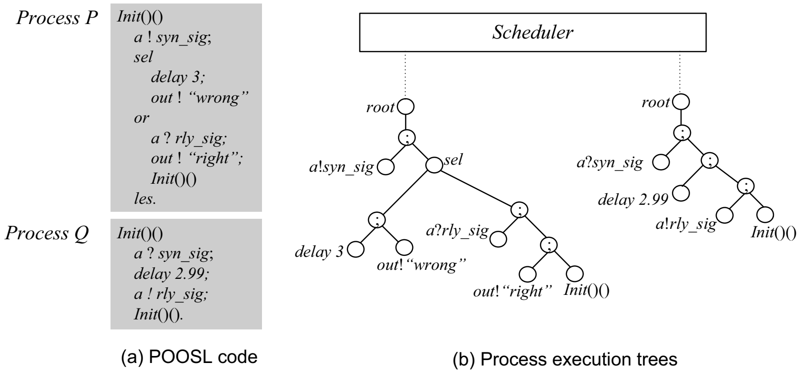
language. Second, the Rotalumis tool is used to automatically synthesis realizations (C++) from POOSL models. Most importantly, the synthesis procedure is based on a formal linkage between between the semantics of the design language (POOSL) and that of the implementation language (C++), which guarantees property-preservation between models and realizations.

3.1 The design language POOSL

In this section, we give a brief overview of the POOSL language (Parallel Object-Oriented Specification Language), which is employed in the SHESim tool and developed at the Eindhoven University of Technology. POOSL language integrates a process part based on a timing and probability extension of CCS and a data part based on a traditional object-oriented language [Voeten et al., 1998]. For example, the system in Example 1 can be modelled by the POOSL code shown in Figure 8.7(a). The expressive power of POOSL enables designers to describe concurrency, distribution, communication, real-time and complex functionality of a system using a single executable model. We have successfully applied it to the modelling and analysis of many industrial systems such as a network processor [Theelen et al., 2003], a microchip manufacture device [Huang et al., 2002] and a multimedia application [van Wijk et al., 2002].

Similar to some other recent design languages equipped with adequate predictability support, the semantics of the POOSL language [van der Putten and Voeten, 1997], [van Bokhoven, 2002] is also based on a two-phase execution model, which guarantees the predictability support during system modelling.

The implementation of the two-phase execution model in simulation tool SHESim is achieved by adopting so-called process execution trees (PETs). The state of each process is represented by a tree structure where each leaf is a statement or a recursively defined process method (an example is the PET of $P \parallel Q$ shown in Figure 8.7(b)). During the evolution of the system, each PET provides its candidate actions to the PET scheduler and dynamically adjusts its state according to the choice made by the PET scheduler. More details about PET can be found in [van Bokhoven, 2002]. The correctness of PETs with



```
Process P   Init()()
              a ! syn_sig;
              sel
                delay 3;
                out ! "wrong"
              or
                a ? rly_sig;
                out ! "right";
                Init()()
              les.

Process Q   Init()()
              a ? syn_sig;
              delay 2.99;
              a ! rly_sig;
              Init()().
```

(a) POOSL code

(b) Process execution trees

Figure 8.7. The $P \mid\mid Q$ system in POOSL

respect to the semantics of the POOSL language is formally proven in [Geilen, 2002].

3.2 Rotalumis

The generation tool Rotalumis takes the POOSL model acquired during the modelling stage as its input and automatically generates the executable code for the target platform. To ensure property-preservation during the transformation, a formal linkage between two semantic domains of the modelling and implementation languages is built based on the ϵ-hypothesis [Huang et al., 2003]. The ϵ-hypothesis requires that:

1 A model and its realization should have the same observable execution sequence.

2 Time deviations between activations of corresponding actions in the model and the realization should be less than ϵ seconds.

In the case that the ϵ-hypothesis is complied with during the transformation, we could predict properties of the realization from those of the model. More specifically, if the model satisfies a property P formally specified by MITL (Metric Interval Temporal Logic)[Alur et al., 1991], we know that the realization satisfies a 2ϵ relaxed property $R^{2\epsilon}(P)$ of P [Huang et al., 2003]. For example, a typical response property that "every input p must be followed by a response q between 3 and 5 time units" is defined by formula $\Box(p \rightarrow \Diamond_{[3,5]}q)$. Its 2ϵ relaxed property is $\Box(p \rightarrow \Diamond_{[3-2\epsilon,5+2\epsilon]}q)$. In case an upper bound of the time deviation between the realization and the model is 0.01 seconds and $\Box(p \rightarrow \Diamond_{[3,5]}q)$ is satisfied in the model, we can conclude that property $\Box(p \rightarrow \Diamond_{[2.98,5.02]}q)$ holds in the realization.

The Rotalumis tool tries to satisfy the hypothesis by applying the following techniques:

Process execution trees. POOSL language provides ample facilities to describe system characteristics such as parallelism, nondeterministic choice, delay and communication that are not directly supported by C++ or other implementation languages. In order to provide a correct and smooth mapping from a POOSL model to a C++ realization, PETs are used to bridge the semantic gap between two languages. The data part of a POOSL model is directly translated into corresponding C++ expressions since no large gap exists between their semantics. The process part of a POOSL model is interpreted as a C++ tree structure whose behavior is the same as the PET implemented in SHESim. As a result, the synthesized realization exhibits exactly the same behavior as that in the model, if we interpret it in the virtual time domain.

On the other hand, the realization of a system needs to interact with the outside world and its behavior has to be interpreted in the physical time domain. Since the progress of the virtual time is monotonically increasing, which is consistent with the progress of the physical time, the event order observed in the virtual time domain should be consistent with that in the physical time domain. That is, the PET scheduler ensures that the realization always has the same event order as observed in the POOSL model. Therefore, any qualitative timing property (such as safety and liveness) satisfied in the model also holds in the realization.

Synchronization between virtual time and physical time. To obtain the same (or similar) quantitative timing behavior in the physical time domain as in the virtual time domain, the PET scheduler tries to synchronize the virtual time and the physical time during execution. This ensures that the execution of the realization is always as close as possible to a trace in the model with regard to the distance between timed state sequences[5].

Due to the physical limitations of the platform, the scheduler may fail to guarantee that the realization is ϵ-close to the model (for some fixed ϵ value). In this case, designers can get the information about the missed actions from the scheduler. Correspondingly, they can either change the model and reduce the computation load at a certain virtual time moment, or replace the target platform with a platform of better performance.

With the aid of the Rotalumis tool, a property-preserving realization of Example 1.2 can be automatically synthesized from a POOSL model.

4. Conclusions

To smoothen the system design process and improve design productivity, the semantics of design languages should provide sufficient support for predictable design. More precisely, two aspects should be supported by the semantics of design languages. 1) The semantics of design languages should support compositionality (and composability), thereby facilitating the design of complex systems. 2) A formal linkage between the semantics of modelling and implementation languages is necessary, which can serve as a basis for automatic system synthesis.

In this paper, we investigate the support of the existing design approaches for the above two aspects. The investigation is carried out in two categories of real-time design approaches: platform-dependent approaches and platform-independent approaches. Platform-dependent approaches adopt the physical time as their basic timing concept, often lack sufficient support to model and analyze complex real-time systems, and predictability is not well supported during system modelling. On the other hand, platform-independent approaches adopt the virtual time as their basic timing concept, which improves predictability during system modelling. But they are often ineffective in system synthesis, due to the large semantic gap between modelling and implementation languages.

To cope with the problems of existing design approaches, a predictable approach is proposed, which has two distinct characteristics. First, the POOSL language is adopted during the modelling stage, the semantics of which provides adequate predictability support for real-time system modelling. Second, the Rotalumis tool is used to automatically synthesis realizations (C++) from POOSL models. Most importantly, the synthesis procedure complies with the ϵ-hypothesis, which ensures that realizations keep the same qualitative and quantitative (up to 2ϵ) timing properties as models. In paper [Huang et al., 2004], a rail-road crossing system is presented, which is designed by applying this approach. The analysis of property-preservation between the model of the rail-road system and its realization is presented in [Florescu et al., 2004].

Notes

[1] Although it is often unrealistic to formalize all the requirements of the desired system in practice, we believe that critical timing and safety requirements should be precisely specified.

[2] Physical time can be considered as machine time here.

[3] Since the timing semantics of process Q is influenced by the underlying platform and other processes running on the same platform, in general it is too ambiguous and (almost) impossible to be accurately illustrated by state diagrams. Figure 8.5(a) only shows a part of the semantics of Q, which is already sufficient to show the deficiencies of platform-dependent semantics.

[4] In a complex concurrent real-time (software) system, the cost can far exceed 0.001 seconds due to frequent context switches between many processes.

[5] A timed state sequence is an execution of a system, in which a time interval is attached to every state. If two timed state sequences are ϵ-neighbouring, they have the same state sequence and the least upper bound of the absolute difference between the left-end points of corresponding intervals is less than or equal to ϵ. For more information, see [Huang et al., 2003].

References

Alur, R. Feder, T. and Henzinger, T.A. (1991). The benefits of relaxing punctuality. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 139–152. ACM Press.

Florescu, O. Voeten, J.M.P. Huang, J. and Corporaal, H. (2004). Error estimation in model-driven development for real-time software. In *In Proceedings of Forum on specification and Design Language, FDL'04*, Lille, France.

Geilen, M.C.W (2002). *Formal Techniques for Verification of Complex Real-time Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands.

Graf, S. (2002). Expression of time and duration constraints in sdl. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, LNCS.

Huang, J. Voeten, J.M.P. and Geilen, M.C.W. (2003). Real-time Property Preservation in Approximations of Timed Systems. In *Proceedings of First ACM & IEEE International Conference on Formal Methods and Models for Codesign*, Mont Saint-Michel, France. IEEE Computer Society Press.

Huang, J. Voeten, J. van der Putten, P.H.A. and Ventevogel, A. (2004). Predictability in real-time system development (2) a case study. In *In Proceedings of Forum on specification and Design Language, FDL'04*, Lille, France.

Huang, J. Voeten, J.P.M. van der Putten, P.H.A. Ventevogel, A. Niesten, R. and van de Maaden, W. (2002). Performance evaluation of complex real-time systems, a case study. In *Proceedings of 3rd workshop on embedded systems*, pages 77–82, Utrecht, the Netherlands.

Koomen, C. J. (1991). *The design of communication systems*, volume 147 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston; London; Dordrecht.

Kotonya, G. and Sommerville, I. (1998). *Requirement Engineering: Processes and Techniques*. John Wiley & Sons, Inc., New York.

Leue, S. (1996). Specifying real-time requirements for sdl specifications - a temporal logic-based approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification*, volume 38 of *IFIP Conference Proceedings*, pages 19–34. Chapman & Hall.

Liu, Z. and Joseph, M. (2001). Verification, refinement and scheduling of real-time programs. *Theoretical Computer Science*, 253(1):119–152.

Milner, Robin (1989). *Communication and Concurrency*. Prentice Hall. ISBN 0-13-114984-9 (Hard) 0-13-115007-3 (Pbk).

Nicollin, X. and Sifakis, J. (1991). An overview and synthesis on timed process algebras. In K. G. Larsen, A.Skou, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification, LNCS 575*, pages 376–398, Alborg, Denmark. Springer-Verlag.

Partee, B., ter Meulen, A., and Wall, R. (1990). *Mathematical Methods in Linguistic*. Kluwer Academic Publishers.

Rational Rose RealTime. `http://www.rational.com/tryit/rosert/index.jsp`.

Selic, B., Gullekson, G., and Ward, P.T. (1994). *Real-time object-oriented modeling*. John Wiley & Sons, Inc.

Sifakis, J. (2001). Modeling real-time systems-challenges and work directions. In *Proceedings of the First International Workshop on Embedded Software*, pages 373–389. Springer-Verlag.

TAU Generation 2. `http://www.taug2.com/`.

Theelen, B.D., Voeten, J.P.M., and Kramer, R.D.J. (2003). Performance Modelling of a Network Processor using POOSL. *Journal of Computer Networks, Special Issue on Network Processors*, 41(5):667–684.

van Bokhoven, L.J. (2002). *Constructive Tool Design for Formal Languages from semantics to executing models*. PhD thesis, Eindhoven University of Technology, The Netherlands.

van der Putten, P.H.A. and Voeten, J.P.M. (1997). *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands.

van Wijk, F.N., Voeten, J.P.M., and ten Berg, A.J.W.M. (2002). An abstract modeling approach towards system-level design-space exploration. In *Proceedings of the Forum on specification and Design Language*, Marseille, France.

Voeten, J.P.M., van der Putten, P.H.A., Geilen, M.C.W., and Stevens, M.P.J. (1998). System Level Modelling for Hardware/Software Systems. In *Proceedings of EUROMICRO'98*, pages 154–161, Los Alamitos, California. IEEE Computer Society Press.

Zadrozny, W. (1994). From compositional to systematic semantics. *Linguistics and Philosophy*, 17:329–342.

Z.100 Annex F1: Formal Description Techniques (FDT)–Specification and Description Language (SDL) (2000). Telecommunication standardization sector of ITU.