Predictability in Real-time System Development
(1) Semantics Support from Development Languages *

Jinfeng Huang, Jeroen Voeten
Faculty of Electrical Engineering Eindhoven University of Technology
5600MB Eindhoven, The Netherlands
E-mail: J.Huang@tue.nl

Andre Ventevogel
TNO Industrial Technology
5600HE Eindhoven, the Netherlands

Abstract

With the increasing complexity of real-time control systems, it is important to have sufficient predictability support for a development approach in order to promote the likelihood of the development success. To achieve this, the abstraction/refinement activities during the development should be performed in an efficient way. In this paper, we first briefly introduce the role of the semantics of languages in system development. Then we investigated how the semantics of development languages can support efficient abstraction/refinement. Subsequently, the insufficient predictability support of existing design approaches for real-time control systems are illustrated by an example. Finally, a predictable development approach for real-time control systems is introduced to overcome this problem.

1 Introduction

With the decreasing price and increasing computational abilities of processors, we have witnessed a widespread use of real-time software in various control applications, such as medical instruments, avionic and flight control, traffic control, telecommunication equipments and consumer electronics. These systems are often characterized by concurrency, strict timing constraints and complex functionalities, which add difficulties to the development of qualified real-time software.

Similar to other complex systems, the development of such systems is an activity consisting of several consecutive phases, e.g. the definition of the requirements and constraints (requirement capture), the exploration of possible solutions in the design space (system design) and the generation of a complete executable implementation which satisfies the desired properties (system construction). Among these, the system design is the most crucial and challenging part. During the system design, designers have to discard faulty design solutions and search for a proper solution to bridge requirements and implementations. Such a searching process often involves multiple steps performed in a parallel or sequential way. For example, during the design stage, the whole system is constructed either by the recursive composition of several separately exploited sub-systems or by the stepwise refinement of an abstract model. To smoothen the design process and improve productivity, a systematic design approach is obligatory, which can guide the design through each step. An effective design approach cannot only facilitate efficient

*This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs, the Technology Foundation STW and the Netherlands Organisation for Applied Scientific Research TNO.
evaluation of design ideas at each step, but also keep the consistency during the whole design process. Furthermore, the final design solution, obtained from consecutive design steps, should be smoothly and efficiently transformed into the implementation.

In the past decades, a number of design approaches have been proposed by both academic institutions and industrial organizations. These approaches differ not only in the varying design schemes proposed, but also the different assumptions made during the design. It would be beneficial to compare the typical approaches and sort out what has been accomplished and what needs to be done. In this paper, we perform these comparisons focusing on design efficiency and consistency. Specifically, the efficiency and consistency are addressed in two aspects: those within each design step and within the transformation from the design solution to the implementation. In general, the support of the above aspects in a design approach is determined by the syntax and semantics of requirement languages, design languages and implementation languages, which are called development languages in this paper.

2 Semantics of development languages

The core of a development methodology are development languages, which provide a set of primitives (or vocabularies) by which developers can express their thoughts and domain knowledge [SGW94]. Semantics of development languages has a direct impact on the thinking pattern of developers and the meaning (semantics) of development outcomes. According to the different abstraction levels of design thoughts, three categories of development languages, requirement, design and implementation languages, are involved in the design process.

2.1 Requirement languages

Requirements express the constraints that are put upon a system, each of which is a property that must be present in an implementation in order to satisfy the specific needs of some real-world application [KS98]. Since requirements written in natural languages often cause ambiguity, formal semantics can be adopted to facilitate precise specification of desired system properties, which can contribute to the verification and validation purposes. During verification, the design solutions can be verified against the formal semantics of requirements (properties). During validation, especially the validation of an implementation against its design outcome, it is possible to predict the correctness of the implementation from properties of the model based on a formal linkage between the semantics of three development languages (requirement, design and implementation languages).

2.2 Design languages

Design is the most challenging and creative activity in the cycle of the system development. During the system design, developers need first to understand thoroughly the requirements, carefully explore the design space and finally devise a design solution. The design outcome serves as the basis for later system construction, the success of which depends to a large extent on the design itself. Design languages have decisive effects on the quality of the design outcome, because their syntax and semantics have a direct impact on the designer's thinking pattern. In addition to possessing adequate expressive power to describe both implementable details and relevant aspects of a system, the semantics of a language should also support efficient abstraction/refinement.
2.2.1 Abstraction/refinement

Generally speaking, getting a thorough and clear picture of a real-time software system in-the-large is far from simple. To enable the building up of the whole system from scratch, two mutually inverse activities, abstraction and refinement, are continuously performed during the design process. As shown in Figure 1, abstraction is the activity that tries to remove (or hide) as much as possible irrelevant information from designers, which improves the comprehensibility of the existing design models and facilitates the evaluation of different design solutions. The major concern of the abstraction activities is to improve the understandability of the design, enabling design decisions to be made. On the other hand, refinement is the activity that tries to add more implementation details to the design model, thereby reducing the gap between the implementation and the design model. The major concern 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.

2.2.2 Efficient abstraction/refinement

A design process can be considered as a set of abstraction/refinement activities, the efficiency of which can significantly affect the design time and financial cost of the design. This is especially evident for large-scale and complex software systems. To reduce design complexity, a complex software system is often seen as consisting of a set of interacting components. In practice, compositionality (or composability) is often regarded as an important characteristic that the semantics of a design language should possess, in order to facilitate efficient abstraction/refinement.

**Compositionality** The well-known principle of compositionality [PtMW90] 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 assist developers to understand the meaning of a complex design description. Consider that a system (or subsystem) is represented by a tree structure, where each leaf is a syntactic primitive and each non-leaf node is a combination rule. Compositionality ensures that each syntax sub-tree can be understood 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 is usually far from simple. We can easily foresee that the interpretation of a syntax tree with hundreds of levels, which is often the case for a complex design description, could easily grow beyond the human’s understanding capability. Therefore, compositionality alone does not promise that the meaning of a recursively composed syntax tree can be understood easily.

However, when compositionality combines with abstraction/refinement, it offers many ben-
benefits to reduce the design complexity and to improve the design efficiency. The compositional semantics divides a complex system into isolated semantic components and ensures the semantic independency of each component in the system. Thus, abstraction/refinement activities of the whole can be achieved by the local abstraction/refinement of each component and the mapping of the corresponding combinators, which boosts the overall design efficiency (see Figure 2).

One example of design languages equipped with compositional semantics is CCS (Calculus of Communicating Systems) [Mil89]. Based on the compositional semantics of CCS, observation equivalence is defined, which states that two descriptions are observation equivalent if and only if both descriptions 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 abstraction are iteratively refined by more detailed and equivalent descriptions. In this way, observation equivalence can effectively assist the efficient abstraction/refinement. More detailed discussion about transformational design approaches can be found in [MR96] [vdPV97a]. Example 1 illustrated how abstraction/refinement can be efficiently carried out in CCS.

Example 1 Suppose system $S$ consists of two components $P$ and $Q$, which is given as follows.

$$P \equiv (a \cdot b) \parallel \overline{b} \cdot b, \quad Q \equiv (c \cdot d) \parallel \overline{c} \cdot c \quad \text{and} \quad S \equiv P \parallel Q$$

The semantics of CCS allows designers to consider the abstraction of $P$ and $Q$ independent of each other. In other words, no matter in what context that $P$ or $Q$ is 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 principle, the compositional semantics together with abstraction/refinement can facilitate both top-down and bottom-up design paradigms. During a top-down design process, the design exploration starts at a very abstract level. The correctness of the initial system can be easily verified with the aid of automatic verification tools. Afterwards, designers refine the initial system from abstraction to details by successive refinement steps. The compositional semantics of a design language ensures that the system can be recursively constructed by a set of independent sub-systems till each of them is implementable. The correctness of the refinement can be independently verified locally [CCH+02]. Such a refinement process is also called property-preserving refinement, because all properties of the original system are satisfied by the newly generated system during the refinement. In this way, the design space can be efficiently explored at different abstraction levels and the design decisions can be made through successive refinement.
In contrast to the top-down design paradigm, the bottom-up design paradigm starts design exploration from a set of detailed subsystems. By exploring different ways to combine subsystems, designers expect to find a design solution satisfying desired properties. In this process, designers need to deal with property verification of a combined system based on its subsystems, which often suffers from the state space exploration problem. Compositional semantics of a language provides an efficient way to do so. First, compositional semantics ensures that the behavior of the individual subsystems are independent. Hence, the verification of desired local properties can be carried out separately. Consequently, the verified properties can be considered as abstractions of the original system. This process can also be considered to be a property-preserving abstraction. In this way, different composed systems can be investigated efficiently based on the abstraction of each constitution without having to consider their detailed internal behaviors.

In summary, suppose a system $S \equiv P_1 \oplus P_2 \ldots \oplus P_n$ expressed by a language with compositional semantics, where $P_1, P_2, \ldots, P_n$ are components of $S$ and $\oplus$ is a combinator of components. The compositional semantics states that the abstraction/refinement $P'_1, P'_2, \ldots, P'_n$ of $P_1, P_2, \ldots, P_n$ can be carried out independently. Therefore an abstraction/refinement of $S$ can be $S' \equiv P'_1 \odot P'_2 \ldots \odot P'_n$, where $\odot$ is the corresponding mapping of combinator $\oplus$ in $S'$. In Example 1, $\oplus$ and $\odot$ are both the parallel composition combinator ($\|\|$). In practice, $S'$ can be expressed using the same language as that used for $S$ or a totally different language. For example, properties of a system written in a requirement language can be abstractions of a system written in a design language.

### 2.2.3 Composability

The concept of compositionality is intuitively useful in facilitating the abstraction/refinement efficiency. However, it is not always effective in practice. A major reason is that it does not put enough restrictions on the meaning assignment of combinators. As a consequence, the semantic independency can always be achieved by assigning a trivial semantics to combinators [Zad94]. In practice, the combinator semantics in both abstractions and refinements should be simple enough. For example, the semantics of the combinators $\|$ and $+$ in CCS is defined in a natural manner and can be understood easily. The abstraction/refinement of sub-processes in CCS retains the original combinators between them.

In the context of concurrent reactive systems, a more restricted “version” of compositionality is sometimes called composability. Composability states that properties satisfied by individual components of the system should be satisfied by their parallel compositions [Sif01]. For example, a reactive system $S$ consists of two parallel components $P$ and $Q$. A timing response property $\varphi$, which states that every environment stimuli $p$ must be followed by a response $q$ within 3 seconds. If $P$ satisfies $\varphi$ and the design language supports composability, then $S \equiv P \| Q$ should also satisfies $\varphi$.

More generally, consider a system $S \equiv P_1 \| P_2 \ldots \| P_n$ expressed by a language supporting composability, where $P_1, P_2, \ldots, P_n$ are components of $S$ and $\|$ is the parallel combinator. Assume each component $P_i$ satisfies property $\varphi_i$ independent of the other components. Composability of a design language states that $S$ satisfies the simple logical conjunction of these individual properties ($\varphi_1 \land \varphi_2 \ldots \land \varphi_n$). We can see that only the parallel operator ($\|$) and the logic conjunction ($\land$) are used in composability, which avoids potential “trivial” combinator in compositionality.

### 2.3 Implementation languages

System construction is an activity to convert the design outcome into a complete and executable system while preserving its correctness. During this stage, the system is often expressed by an implementation language (such as C and C++), the semantics of which is usually related with
and constrained by the underlying platform. Due to the gap between two different semantic domains adopted in the design language and the implementation language, the correctness of the design outcome is difficult to ensure in the final implementation by manual interpretations of developers.

The difficulty of maintaining correctness between the design outcome and its implementation is attributed to several reasons. First, certain assumptions are often taken on the semantics of design languages in order to efficiently explore the design space, such as instantaneous actions for timing behaviors [NS91]. These assumptions on semantics are valid at a certain level of abstraction, but they do not always hold for the semantics of implementation languages. For example, every action does take certain amount of execution time in all implementation languages. Second, some primitives and operations in design languages do not have direct correspondence in implementation languages. For example, during system construction, the default concurrency operation is often represented by a specific thread mechanism offered by the underlying operating system, whose semantics is not always consistent with that in design languages. The study on property preservation between design and implementation phases tries to build a formal linkage between two different semantic domains facilitating automatic and correct system generation[HVG03][HVVvB04].

The remainder of the paper is organized in four sections. In Section 3, we will briefly explain the deficiency of existing approaches in supporting predictability during the development of real-time control systems. To solve the problem presented in Section 3, we introduce an approach which provides adequate predictability support for real-time control system development in section 4. Section 5 gives conclusions and future work.

## 3 Real-time system design approaches

In this section, we are going to evaluate whether existing design approaches have adequate semantic support for real-time software systems. We classify existing design approaches into two categories, platform-dependent design and platform-independent design, based on the different timing concepts adopted in these approaches. This is a justified classification because approaches adopting the same timing concept often exhibit similar characteristics. Briefly speaking, platform-independent approaches take use of a system variable to represent time (denoted as the virtual time in the thesis), while platform-dependent approaches often adopt the physical time to specify the timing behavior of the system.

### 3.1 Platform-dependent design: Inefficient abstraction/refinement

The timing semantics of the design languages in platform-dependent approaches is often too ambiguous to support efficient abstraction/refinement. This can be illustrated by Example 2.

**Example 2 Two synchronized processes P and Q:** Consider a simple real-time system (shown in Figure 3) consisting of two parallel processes P and Q (P || 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 3 seconds delay and process Q sets a timer with 2.999 seconds 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”.

---

1^The development of a railroad crossing system using the proposed approach is illustrated the companion paper [HVvdPV04]
in the integration. Often reasoned from their abstractions. This would indicate that properties of the integration of their abstractions. Therefore, the integrated system (design practice, it is often assumed that the integration of parallel processes can preserve the

\[ Q \]

As a consequence, Figure 4b can be considered to be a correct abstraction of process \( Q \). Similarly, an abstraction can be obtained for process \( P \), which is depicted in Figure 4c. In

diagrams. Figure 4a only shows a part of the semantics of \( Q \). This part of the semantics is already sufficient enough to show the deficiencies of platform-dependent semantics.

---

The process first receives a message \( \text{syn}_\text{sig} \), which takes time duration \( t_1 \). Before the next statement (\( \text{set}(\text{qtimer, now} + 2.999) \)) 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 \text{time\_out} message, again the operating system might take a total amount of time \( t_4 \) for the execution of other processes. The response to the \text{time\_out} message takes time duration \( t_5 \).

In reality, execution time \( t_1, t_3, t_5 \) and \( t_6 \) are negligible for modern computer platforms. 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 4b can be considered to be a correct abstraction of process \( Q \). Similarly, an abstraction can be obtained for process \( P \), which is depicted in Figure 4c. In design practice, it is often assumed that 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 from their abstractions. This would indicate that \( P \) should never reach state \( S'_5 \) in the integration.

---

2Since the timing semantics of process \( Q \) is influenced by the underlying platform and other processes in the same platform, in general it is too ambiguous and (almost) impossible to be accurately illustrated by state diagrams. Figure 4a only shows a part of the semantics of \( Q \). This part of the semantics is already sufficient enough to show the deficiencies of platform-dependent semantics.

---

Here, we use a graphical design language based on SDL-96 to describe the system (shown in Figure 3). In SDL-96, the timing semantics is given in such a way that each action takes an undefined amount of physical time \([z1099][Gra02]\) and the interpretation of timing expressions (such as timers) relies on an asynchronous timer mechanism provided by the underlying platform [Leu96].
However, in certain circumstances, the platform-dependent semantics of both processes does allow process $P$ to reach state $S'_5$ in the integrated system. For example, in Figure 4a, when process $Q$ is in state $S_1$, the underlying operating system can first switch the active process to $P$, where $P$ sets the timer and suspends itself, then switches back to $Q$ to set a timer with the duration of $2.999$ seconds. If one context switch, one timer setting, one process suspending and other necessary scheduling execution take more than $0.001$ seconds in total $^3$, the timer of process $P$ might expire before that of process $Q$. As a result, $P$ outputs a message “wrong”.

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

Two techniques are often adopted in practice to alleviate the problems mentioned above for platform-dependent design approaches.

**Atomic constructs:** The ambiguity of the semantics is mainly caused by the scheduling strategies of the underlying operating system, which may stop abruptly the current process and switch to actions in other processes. To avoid the interference from other processes during the execution of a process, atomic constructs are often adopted in design languages [HF03], which group a set of actions and consider them as one that cannot be interrupted by other processes. In atomic constructs, the computation time is not efficiently used, because it is forced to be assigned to a single process for a certain period of time, even when the process only makes use of a minor fraction of that allocated time. Furthermore, the exclusive assignment of the computation time limits the expressive power of the language, especially in dealing with concurrent timing behaviors mapping on a single processor.

**Real-time scheduling:** In the task-level real-time scheduling research domain, 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. The scheduler 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 task-level 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, task-level scheduling lacks a consistent framework to integrate functionality and timing [LJ01] and it is often ineffective for interaction-intensive real-time applications.

### 3.2 Platform-independent design: ineffective system construction

In the previous subsection, we have seen platform-dependent approaches are ineffective in the design and analysis of complex real-time systems. Recently, platform-independent approaches, such as SDL-2000 [z1000] adopting virtual time concepts in their design languages has been proposed. The semantics of design languages in platform-independent approaches often assume that system actions (such as communication, data computation) are timeless (taking zero time) and time passes without any action being performed. On one hand, such semantics provides sufficient expressive power to describe the real-time behavior of a control system. On the other hand, compositionality and composability can be supported in this semantic framework. A typical design language based on this semantic framework is SDL-2000 [z1000], which is supported

---

$^3$In a complex concurrent real-time software system, the cost can far exceed 0.001 seconds due to frequent context switches between processes.
The timing semantics of the design descriptions in Example 1 can be defined in this semantic framework. Accordingly, \( t_1 \) to \( t_6 \) are all zero. Then, we can safely say that the semantics depicted in Figure 4a can always be abstracted correctly to that in Figure 4b and the abstraction of process \( P \) can be that shown in Figure 4c. Consequently, the semantics of the combined system \( P \parallel Q \) can be efficiently abstracted as the combination of Figure 4b and Figure 4c, in which process \( P \) should never reach state \( S'_5 \). We made the same model in TAU G2, and the behavior of the system \( (P \parallel Q) \) is indeed as we expected.

![Figure 5: The implementation of \( P \parallel Q \)](image)

Although sufficient predictability support has been provided in the design languages of platform-independent design approaches, these approaches still lack a formal linkage from the semantics of its design language to that of its implementation language, which has been explained in Section 2.3. For example, in TAU G2, a function is provided to automatically transform design descriptions into corresponding implementation code. The transformation is achieved mainly by the syntactic mapping of syntax primitives and constructs between two development languages, instead of by a semantic mapping. Figure 5 shows the output of an automatically generated implementation of Example 1 by TAU G2. We can see that the implementation exhibits “unexpected” behavior, sending out a “wrong” message. More examples given in [HVVvB04] demonstrate the inconsistency between a design description and its realization.

### 4 A predictable development approach

In the previous section, we have investigated the deficiency of the existing design approaches in supporting predictability for real-time control systems. In this section we introduce a development approach which can overcome this problem. This approach has two distinct characteristics. First, during the design stage, The POOSL language is adopted, the semantics of which provides adequate predictability support for the design process. Second, tool Rotalumis is used to automatically construct implementation (C++) from the POOSL model. Most importantly, the construction 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-preserving system construction.

#### 4.1 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 [VvdPGS98]. For example, the system in Example 1 can be modelled by the POOSL code shown in Figure 6a. The expressive power of POOSL language enables designers to describe concurrency, distribution, communication, real-time and complex functionality features of a system using a single executable model. We have successfully applied it to the modelling and analysis of many industrial systems such as an internet router [TVvB*01], a network processor [TVK03], a microchip manufacture device [HVvdP*02] and a multimedia application [vWvTB02].

by TAU G2 released by Telelogic [TAU].
Similar to some other recent design languages equipped with adequate predictability support, the semantics of the POOSL language [vdPV97b, vB02] is also based on a two-phase execution model. In this execution model, the state of a system can change either by asynchronously executing some atomic actions such as communication and data computation without time passing (phase 1) or by letting time pass synchronously without any action being performed (phase 2). Different from traditional design languages, the time progress of a system is recorded by a variable (virtual time) instead of using by physical time. Besides providing effective semantic equivalence relation for design descriptions, this timing mechanism also brings a lot of other benefits [HVVvB04] to system design. For example, the timing behavior of the system is interpreted uniquely and is not “polluted” by extra debugging and analysis code.

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 6b). 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 [vB02]. The correctness of PETs with respect to the semantics of the POOSL language is formally proven in [Gei02].

4.2 Rotalumis

The generation tool Rotalumis takes the POOSL model acquired during the design stage as its input and automatically generates the executable code for the target platform. To facilitate the property-preservation, a formal linkage between two semantic domains (design and implementation) is built based on the $\epsilon$-hypothesis, which guarantees that the implementation keeps the same qualitative and similar quantitative timing properties as in the model [HVG03]. The $\epsilon$-hypothesis requires that:

1. The implementation and the model should have the same observable execution sequence.
2. The time deviation between activations of the corresponding actions in the implementation and the model should be less than or equal to $\epsilon$ seconds.

In the case that the $\epsilon$-hypothesis is complied with during the transformation, we could predict properties of the implementation from those of the model. More specifically, if the model satisfies a property $P$ formally specified by MITL (Metric Interval Temporal Logic)[AFH91], we know that the implementation satisfies a $2\epsilon$ relaxed property $R^{2\epsilon}(P)$ of $P$ [HVG03]. 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 $2\epsilon$ relaxed property is $\Box (p \rightarrow \Diamond [3-2\epsilon,5+2\epsilon]q)$. In the case that an upper bound of the time deviation between the implementation 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 implementation.

The $\epsilon$-hypothesis is incorporated into the generation tool Rotalumis by applying the following techniques:

1. **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++ implementation, 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 generated implementation exhibits exactly the same behavior as that in the model, if we interpret it in the virtual time domain.

   On the other hand, the implementation 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 monotonic 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 scheduler of PET ensures that the implementation 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 implementation.

2. **Synchronization between virtual time and physical time:**

   To obtain the same (or similar) quantitative timing behavior in the physical time domain as in the model, the scheduler of PETs tries to synchronize the virtual time and the physical time during the running of the implementation, which ensures that the execution of the implementation is always as close as possible to a trace in the model with regard to the distance between timed state sequences.

   Due to the physical limitation of the platform, the scheduler may fail to guarantee that the implementation is $\epsilon$-close to the model, for a fixed $\epsilon$ 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 cost of 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 implementation of Example 1 can be automatically generated from a POOSL model. In the companion paper, a more complex case (rail-road crossing system) is carried out by applying this development approach.

5 Conclusions

To smoothen the system development process and improve development productivity, the semantics of development languages should provide sufficient support for efficient and consistent design.

---

4 A timed state sequence is an execution of the system in which a time interval is attached to every state. If two timed state sequences are $\epsilon$-neighbouring, they have exactly the same state sequence and the least upper bound of the absolute difference between the left-end points of the corresponding intervals is less than or equal to $\epsilon$. For more information, see [HVG03].
More precisely, two aspects should be supported by the semantics of development languages. 1) The semantics of design languages should support compositionality (and composability), thereby facilitating the efficient design of complex systems. 2) A formal linkage between the semantics of design languages and implementation languages is necessary, which can serve as a basis for automatic system generation.

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 physical time as their basic time concept, and often lack sufficient support for design and analysis of complex real-time systems. On the other hand, platform-independent approaches adopt virtual time as their basic time concept, and improve the efficiency and consistency of the design of complex real-time systems. But they are often ineffective in the system generation, due the semantic gap between design languages 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 design stage, the semantics of which provides adequate predictability support for real-time control system design. Second, the Rotalumis tool is used to automatically construct implementations (C++) from the POOSL model. Most importantly, the construction procedure is based on the ϵ-hypothesis which ensures that the implementation keeps the same qualitative properties and similar quantitative timing properties as the model. In the companion paper [HVvdPV04], a rail-road crossing system is presented, which is developed by applying this approach.

References


