Towards correctness-preserving synthesis for real-time software*

Jinfeng Huang[1], Jeroen Voeten[1], Andre Ventevogel[2], Leo van Bokhoven[3]
[1]Eindhoven University of Technology, Faculty of Electrical Engineering
E-mail: J.Huang@tue.nl
[2]TNO Industrial Technology, 5600HE Eindhoven, the Netherlands
[3]Magma Design Automation B.V. 5600MB Eindhoven, The Netherlands

Abstract—

It is well-known that real-time systems (especially real-time software) impose a lot of challenges to system design and implementation. In this paper, we first analyze the difficulties the design problems stem from. Then we investigate the merits and pitfalls of major design styles by evaluating them against predictability, compositionality, efficiency, expressive power and debuggability. Finally we propose a design approach which combines both the time-driven and event-driven design styles for real-time software development. The proposed approach consists two consequent procedures:

1. Platform-independent design provides a virtual execution environment in which the design description is uniquely interpreted and formally evaluated by verification and simulation techniques without being affected by non-deterministic factors of the underlying platform. Base on the analysis results, the model can be elaborated smoothly to satisfy all essential functional and critical timing requirements.

2. Correctness-preserving generation takes the model as input and automatically generates completely executable software. Such a generation complies with the ϵ-hypothesis [1], which guarantees the generated implementation from model keeping the same qualitative (such as the order of actions) and similar quantitative (such as deadline) timing properties.

This design approach not only has the flexibility of the event-driven design style for modelling complex real-time systems but also keeps predictability and compositionality merits of the time-driven design style during the development.

I. Introduction

Real-time softwares have been in widespread use in various applications such as medical instruments, missile control systems, automobile control systems, multimedia systems and traffic control systems. These systems often exhibit complex behavior because of intricate interactions between their components, while real-time requirements add another dimension to the complexity of a system. One common feature of these systems is that their correctness is determined by both the system outputs and their issued time. These two aspects are closely related to each other and either one of them could exert great influence on the other. For example, the same interaction between components issued at different time might produce totally different outputs. In practice, the design of such a complex real-time software becomes extremely difficult because proper methods are still lacking to integrate both aspects (time and functionality) seamlessly and keep these real-time interactions under control.

During the development, to ensure the predictability and analysability of the timing behavior of the real-time software which consists of a set of components, time and functionality should be integrated in the following way.

- Ideally, the timing behaviour of each component is deterministic. For the same timed interaction, the component always produces the same output at the same time point. However, in practice, it is also acceptable that the same output has a small deviation in its issued time for the same timed interaction (pseudo-deterministic).
- The composition of components does not change their own timing properties. Furthermore, the timing behaviour of the composition result should also be pseudo-deterministic.

Some synchronous design approaches (such as Esterel) have enjoyed a lot of success on the consistent integration of time and functionality during the system development. At the design stage, duration of each action is treated as instantaneous and time passing is measured by counting the number of events. In this way, the timing behaviour of each component is deterministic and do not affected by composition with other components, the properties of the whole system can be easily analysed based on the timing properties of each component. The implementation can

*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.

be automatically generated from the model by complying with the digital synchrony hypothesis, which assumes that each action is performed within 1 digital clock and simultaneous actions are synchronized within 1 digital clock. Due to the fact that digital synchrony hypothesis is not far from the perfect synchrony hypothesis made in the model, the generated implementation has almost the same qualitative and quantitative timing properties as the model [2] [3]. These approaches have achieved fruitful results in the area of reactive real-time system development.

However, the same approaches can not easily be brought into the design of complex interactive real-time software systems. These systems often run on a certain platform containing hardware and software such as operating systems and middleware. The difficulties are mainly caused by nondeterministic factors introduced by those techniques, such as caches, pipelines, instruction pre-fetch technique, memory management and process scheduling, which have been widely applied in HW/SW platforms. These techniques, on one hand boost the overall computation performance and enhance the flexibility of the underlying platform, on the other hand severely hamper developers to make an adequate estimation of the time cost of a certain functionality running under the platform. A typical real-time software design flow often separates two aspects (time and functionality) and starts from the latter with the assumption of zero or a fixed nonzero time cost. The actual time cost of a certain functionality is determined at run-time and can be affected by the nondeterministic factors of the underlying platform. Since usually no proper measures are taken to prevent such inconsistency on the time cost on changing the timing properties of the system, during the development, developers have to rely on manual adjustment of the their functionality design to eliminate the effect of the timing inconsistency. However, this has been proven inefficient and often infeasible in practice.

II. All about time

A. Time – a shared resource

In principle, resources can be considered as entities that have limited quantity and are used by one or more consumers [4], for instance, memory, CPU cycles and bandwidth. If a resource can be shared by more than one consumers, a scheduler (or resource manager) is needed to solve the usage conflictions between consumers and to improve the usage efficiency of the resource.

Time is not thought to be a resource in non-real-time systems, simply because there is no limitation to the usage of time and the systems can consume as much time as they need. However, in real-time systems, qualitative and quantitative timing constraints are put on system actions, which requires that actions are performed in certain periods of time and actions from different tasks have to be executed in an expected order in time. Due to the finite computational ability of the platform, computation time is limited. In the case that the limited computation time has to be shared by several tasks, a scheduler is required to manage the computation time assignment on them. Furthermore, different from other kinds of physical resources such as memory and bandwidth, we have no control over time. We cannot reuse, stop, stretch or shrink it to fit our needs, which brings a lot difficulties for developing real-time systems [5]. For example, computation time assigned for actions earlier or later than required may not yield a valid result, and sometime may lead to disasters.

B. Scheduler– time resource manager

A time scheduler in real-time systems must keep track of all available timing actions and assign the computation time to the most urgent actions. Furthermore, the scheduler should also includes timing exception handler to notify designers, in the cases that the computation time is not enough to satisfy all urgent actions. The goal of the scheduler is to optimize the usage of the limited computation time without undue time cost. An ideal scheduler would spend negligible time load and reasonable memory consumption.

The benefits which can be provided by such a scheduler are introduced as follows:

1. Real-time software design can be platform-independent. Designers only need to specify the expected issuing time or deadline for actions in the design. The timing behaviour of the design is deterministic and is not affected by the non-deterministic factors of the underlying platform. The scheduler takes care that the execution of actions is timely during run-time. It relieves the burden for designers who have to manually adjust their design to eliminate (or reduce) the influence of non-deterministic factors.
2. Compositionality can be better supported during the design stage. It is widely known that traditional real-time design approaches are not friendly with compositionality. When removing or adding a functional component into the whole software system, the timing behaviour of other components may be influenced dramatically. However, in the case that a global scheduler takes care of the execution of each action from different components, actions are scheduled according to their timing requirements without the interfere with other components. Timing properties of each component are not changed during the composition of components.
3. Real-time exception can help designers locate their er-

rors and guide the refinement process of the design. A real-time exception is thrown when the computation time resource does not suffice to meet the deadline of all timing actions. In this case, designers can either refine their design to reduce the time congestion or choose a faster platform which could provide more computation time for the software.

4. Computation time can be used in a more efficient way. Since the scheduler has the global information about timing constraints of all available actions from different tasks (or components) in the system, it could optimize the usage of computation time to meet all timing constraints at the same time to keep the timing behaviour of each task (or component) predictable.

To summarize, a scheduler plays an important role in the integration of functionality and time aspects of the system and keeps the timing interactions between different components under control during the whole development. In the next section, we will present several real-time styles and investigate how well those approaches combine time resource and functionality together to develop a correct real-time software.

III. Real-time Design Styles

In this section we present several design styles which have been used to develop real-time systems. Each design style has its own design principles which are based on certain assumptions. In the sequel, we evaluate the ability of each style to integrate time and functionality effectively.

Before the introduction of these approaches, we first present several aspects from which our evaluation results are derived.

- **Predictability of the target system behaviour:** to guarantee the timeliness of a real-time system, we expect that its timing behaviour is predictable and all critical timing constraints are ensured during the development.
- **Compositionality of the design:** a complex system is often decomposed into sub-components progressively until each individual component can be easily understood and implemented. Compositionality requires that the composition of these individual components does not change their own properties. In the context of the real-time system development, compositionality means that timing properties of each individual component are preserved, and in addition, the timing behaviour of the composition result is still pseudo-deterministic. Compositionality is an essential way to reduce the design complexity of the system.
- **Scheduling efficiency:** a complex real-time system usually consists of a set of concurrent tasks. The context switch between tasks is unavoidable in most of these systems, which consumes additional time resources. Too much switch cost will restrict the usability and applicability of a design approach.
- **Expressiveness:** a complex real-time system often has features of timeliness, concurrency and complex functionality. To express these features during the development, a design approach should be equipped with adequate expression power to describe timing requirements, concurrency, communication, the system's structure and data types.
- **Debuggability:** additional debug codes may change the behaviour of the system and introduce new errors (Heisenberg principle in testing [6] [7]). This is especially evident in the debugging of real-time systems. Without careful considerations and treatments, debug codes can easily "pollute" the original timing behaviour of the system and interfere with the localization and correction of errors.

A. Real-time scheduling

Computation systems usually have multiple tasks running concurrently. In real-time systems, all the eligible tasks have to compete for the limited computation time. Therefore, a scheduler is usually needed to manage the activation and execution of each task. The scheduler assigns computation time by giving different priorities to tasks. In general, the task with a higher priority is scheduled before those tasks 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. However, it is well-known that the general scheduling problem is NP-complete and computationally intractable [8]. In order to reduce the complexity of the design of a feasible scheduler, designers have to rely on a set of assumptions such as fixed execution time, independent tasks, fixed priorities, no shared resource or periodic task triggers.

Furthermore, following the priority assignment blindly and triggering the task with the highest priority available may cause the priority inversion problem, in which a task with a higher priority can be blocked by tasks with lower priorities. For example, a system is running A, B and C tasks. Task A has the highest priority, B has normal priority and C has the lowest priority. If A needs to synchronize with C, it has to wait until B finishes, because B has a higher priority than C and is scheduled before C. The priority inversion problem can be solved by priority inheritance and priority ceiling protocols which, however, may introduce significant run-time overhead and affect the predictability of the system.

Although real-time scheduling is one of the most important analytical methods in real-time system design, it is still not a complete solution for developing predictable real-time systems, whose pitfalls can be illustrated by the following aspects.

1. The timing behaviour of the system is sensitive to the activation time and the actual execution time of each task. The predicability of the system behaviour might be broken by an earlier arrival of a higher priority task or an unexpected longer execution [9].
2. The scheduling algorithm can be easily affected by the the number of tasks and the execution time of each task. Modifying, removing or adding a task often lead to the redesign of the scheduler of the system.
3. Context switch between tasks and the execution of sophisticated scheduling algorithm is time-consuming. Furthermore, the scheduling algorithm is devised according to the worst-case execution time which is often much longer than the actual execution time of tasks. Hence, time resource can not be efficiently used for the execution of real-time systems.
4. In the design of complex real-time systems, tt is often difficult to keep those assumptions which are used to reduce the complexity of the design of a scheduler.

B. Time-driven design style

The time-driven design style usually treats time as the only trigger for computation in the system. Timing constraints are specified in the programming model and are used to guide the scheduling of run-time tasks. In this way, the deterministic behaviour of the system can be achieved. The communication between tasks is usually defined to use a global data space through shared protected data objects and is often extended to use a message passing mechanism [10]. For example, in the Giotto model [11], both computation and communication are triggered by a global time [1]. To accomplish this, the starting and ending time of each task is predefined and communication between tasks can only occur at the beginning or at the end of the task execution. The advantages of time-driven design style include:

1. Predictable system timing behaviour is guaranteed during the development.
2. Compositionality is well supported by assuming the independency of components and time-triggered communication.
3. The scheduler is efficient because the order of task execution is often fixed during the design.
4. Design errors can be easily detected at run-time.

However, the nature of the time-driven style is only suitable for designing those real-time systems with periodical inputs. The task model of the system is relatively simple and complex task behaviours can hardly be described [10].

[1] In a distributed system, all clocks have to be synchronized with a known precision.

C. Event-driven design style

Different from the time-driven design style, all activities are triggered by events in the event-driven style. Each event is associated with a piece of code (event-handler). compared with the object-oriented technique, event-driven design style has more expressive power to model complex system behaviours. However, there is no explicit timing constraint put on the execution of event-handler and interactions between components of the system, As a result, the timing behavior of the system can be easily affected by many non-deterministic factors of the undering platform and is hard to analyze and predict. Design errors of the system are often hard to trace because debug codes might change the timing behaviour severely.

From the above evaluation, we can see that each design style has its merits and pitfalls for developing real-time software systems. In the next section we will present a new design style which can be considered as a combination of the time-driven design style and the event-driven design style.

IV. A COMBINATION OF EVENT-DRIVEN AND TIME-DRIVEN DESIGN STYLES

In many formal mathematical frameworks for reasoning concurrent real-time systems, a model of the system is structured by the way of the event-driven style. The formal model consists of a set of concurrent processes which are asynchronously executed and they synchronously or asynchronously communicate with each other. The behaviour of each process is largely determined by its interactions with other processes. Different from event-driven design style, here actions in event-handler codes of each event are abstracted as instantaneous and triggered by time, which ensures the predictability of the timing behaviour of each process. Formal design languages based on these mathematical frameworks often integrate object-oriented techniques, which gives adequate expression power and flexibility to real-time system design without losing predictability and analyability of the timing behaviour of the system. However, these mathematical frameworks usually assume instantaneous actions, which are not always satisfied by the actual execution environment. As a consequence, the generated implementation from the formal model is difficult to guarantee the predictability of the timing behaviour and preserve the same quantitative and similar qualitative timing properties. In the sequel, we present a design approach which seamlessly combines formal design method with correctness-preserving generation. The approach inherits the merits of both the time-driven design style and the event-driven design style. It consists of two

consequent procedures, platform-independent design and correctness-preserving generation.

A. Platform-independent design

The core of platform-independent design is an expressive and well-founded modelling language, which provides ample means for making adequate system models and at the same time gives unambiguous interpretation on the model. POOSL is one of such languages which integrates a process part based on timed CCS and a data part based on a traditional object-oriented language [12]. The expressive power of POOSL language enables designers to describe concurrency, distribution, communication, real-time and complex functionality features of a system in a single executable model. The formal semantics behind the POOSL language gives an unique interpretation on the model which can be further simulated, analysed and verified in a virtual execution environment. Thus timing properties of the system can be formally evaluated by verification and simulation techniques without being affected by non-deterministic factors of the specific underlying platforms. Base on the analysis results the model can be elaborated smoothly to satisfy all essential functional and critical timing requirements.

B. Correctness-preserving generation

The generation tool takes the model as its input and automatically generates the completely executable software, To guarantee that the generated implementation from model keeps the same qualitative and similar quantitative timing properties, in our approach, the ϵ-hypothesis has to be complied with during the run-time execution [1]. The ϵ-hypothesis requires that:
1. The implementation and the model should have the same observable execution sequence.
2. Time between activations of the corresponding actions in the implementation and the model should be less than or equal to ϵ seconds.
The ϵ-hypothesis is incorporated into the generation tool Rotalumis by applying the following techniques:
1. Execution trees are used to bridge the gap between the expressibility difference between POOSL and C++ language. POOSL provides ample facilities to describe system characteristics such as parallelism, preemption, non-deterministic choice, delay and communication that are not directly supported by C++. In order to provide a correct and smooth mapping, execution trees are adopted to represent individual processes of the model and a scheduler calculates their next step actions in the execution of the model. The correctness of this execution method with respect to the semantics of the POOSL model has been proven in [13]. Therefore, the generated C++ software implementation will always have the same (untimed) event order as observed in the POOSL model. More details about the execution trees can be found in [14].
2. In the implementation, the scheduler of execution trees tries to synchronize the virtual time in the model and the physical time in 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 [16]. Due to the physical limitation of the platform, the scheduler may fail to guarantee the timing constraints specified in the model, even with ϵ-relaxation. In this case, designers can get the information about the missed actions. Correspondingly, they can either refine the model and reduce computation cost, or replace the target platform with a platform of better performance.

C. Evaluation

This section evaluates our design approach based on those aspects we have proposed before.
- At the design stage, the execution semantics of the model has two alternative phases [15]. 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). In this way, the timing behaviour of a model is always predictable with respect to the virtual time. Since actions are instantaneous in the model, the composition of different components still keeps their own timing properties and has deterministic timing behaviours. In the actual run-time system, predictability and compositionality of the system is ensured by the ϵ-hypothesis [16].
- The scheduler and concurrent activities are multiplexed on a single process. The switch between concurrent activities is accomplished by performing application-controlled context switching between actions. These require less context information to be saved, which is much more efficient than the process level context switching. Furthermore, with the optimization, the switch times between actions can be almost the same as that of the process level context switching in other design styles.
- It has adequate expressive power to describe complex real-time systems. The approach has been successfully applied to model and analyse many industrial systems such as an internet router[17], a network processor[18] a microchip manufacture device[19] and a multimedia application[20].
- Two different debug mechanisms have been provided during the development. At the design stage, since actions

are instantaneous and extra debugging codes do not consume computation time, the time behaviour of the system is kept consistent during the debugging. At run time, the scheduler can detect the failure of the compliation with the ϵ-hypothesis and throw a timing error-hander to inform designers of platform-related timing errors.

Although this design approach has the merits of both the event-driven and the time-driven design styles, its ϵ-hypothesis is not always easily satisfied in data-flow oriented applications, which often involves "heavy" data computations. Further research is still needed to address this issue.

V. Conclusion

In this paper, we have analysed the difficulties experienced when designing complex real-time software. By investigating the merits and pitfalls of major design styles, we propose a design approach which combines the event-driven and time-driven design style. Such a design approach integrates the merits of both event-driven and time-driven design styles, and provides designers with more flexibility without losing predictability and compositionality for developing real-time systems. Initial experiments have been performed that confirm the advantages of this approach [1].

References

[1] J. Huang, J. Voeten, A. Ventevogel, and L. van Bokhoven, "Platform-independent design for embedded real-time systems," in *Proceedings of Forum on specification and Design Languages*, Frankfurt, Germany, Sep 2003.

[2] G. Berry, "A hardware implementation of pure esterel," in *Academy Proceedings in Engineering Sciences*, vol. 17. Indian Academy of Sciences, 1992, pp. 95–130.

[3] ——, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000, ch. The Foundations of Esterel, pp. 425–454.

[4] B. Selic and L. Motus, "Using models in real-time software design," *IEEE Control Systems Magazine, special issue on Advances in Software Enabled Control*, vol. 23, no. 3, pp. 31–42, June 2003.

[5] B. Selic, "Turning clockwise: using uml in the real-time domain," *Communications of the ACM*, vol. 42, no. 10, pp. 46–54, 1999.

[6] J. B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture: 1st edition*. John Wiley & Sons, Inc., Oct 1996.

[7] H. Vranken, *Design for test and debug in hardware/software systems*. Eindhoven : Technische Universiteit Eindhoven, 1998.

[8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers, 1997.

[9] J. Liu and Edward A. Lee, "Timed Multitasking for Real-Time Embedded Software," *IEEE Control Systems Magazine, special issue on Advances in Software Enabled Control*, vol. 23, no. 1, pp. 65–75, Feb 2003.

[10] M. Saksena and B. Selic, "Real-time software design: State of the art and future challenges," IEEE Candian Review, Tech. Rep., 1999.

[11] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree, "From Control Models to Real-Time Code Using Giotto," *IEEE Control Systems Magazine, special issue on Advances in Software Enabled Control*, vol. 23, no. 1, pp. 50–64, Feb 2003.

[12] J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, and M.P.J. Stevens, "System Level Modelling for Hardware/Software Systems," in *Proceedings of EUROMICRO'98*. Los Alamitos, California: IEEE Computer Society Press, 1998, pp. 154–161.

[13] M.C.W Geilen, "Formal techniques for verification of complex real-time systems," Ph.D. dissertation, Eindhoven University of Technology, The Netherlands, 2002.

[14] L.J. van Bokhoven, "Constructive tool design for formal languages from semantics to executing models," Ph.D. dissertation, Eindhoven University of Technology, The Netherlands, 2002.

[15] X. Nicollin and J. Sifakis, "An overview and synthesis on timed process algebras," in *Proceedings of the 3rd workshop on Computer-Aided Verification*, K. G. Larsen, Ed., Alborg, Denmark, July 1991.

[16] J. Huang, J. Voeten, and M. Geilen, "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, June 2003.

[17] B.D. Theelen, J.P.M. Voeten, L.J. van Bokhoven, P.H.A. van der Putten, G.G. de Jong, and A.M.M. Niemegeers, "Performance modeling in the large: A case study," in *Proceedings of the European Simulation Symposium*, Ghent (Belgium), October 2001, pp. 174–181.

[18] B.D. Theelen, J.P.M. Voeten, and R.D.J. Kramer, "Performance Modelling of a Network Processor using POOSL," *Journal of Computer Networks, Special Issue on Network Processors*, vol. 41, no. 5, pp. 667–684, April 2003.

[19] J. Huang, J. Voeten, P. van der Putten, A. Ventevogel, R. Niesten, and W. van de Maaden, "Performance evaluation of complex real-time systems, a case study," in *Proceedings of 3rd workshop on embedded systems*, Utrecht, the Netherlands, Oct 2002, pp. 77–82.

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