Incorporating Time in the Modeling of Hardware and Software Systems: Concepts, Paradigms, and Paradoxes

Martin Ouimet and Kristina Lundqvist
Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
{mouimet, kristina}@mit.edu

Abstract

In this paper, we present some of the issues encountered when trying to apply model-driven approaches to the engineering of real-time systems. In real-time systems, quantitative values of time, as reflected through the duration of actions, are central to the system's correctness. We review basic time concepts and explain how time is handled in different modeling languages. We expose the inherent paradox of incorporating quantitative time-dependent behavior in high-level models. High-level models are typically built before the system is implemented, which makes quantitative time metrics difficult to predict since these metrics depend heavily on implementation details. We provide some possible answers to this paradox and explain how the Timed Abstract State Machine (TASM) language helps address some of these issues.

1 Introduction

The use of models is becoming increasingly popular in the engineering of complex hardware and software systems. The increase in popularity can be partly attributed to the ability to uncover defects during the early phases of the engineering lifecycle, when defects are much cheaper to fix [5]. The ability to model systems and to analyze models gives insight into the potential behavior of the system before it is built [27]. Real-time systems [8] are a special class of computer systems where time plays a critical role in the functionality of the system. Correctness of a real-time system is defined not only in terms of functional correctness but also in terms of timing correctness. A real-time system needs to provide the right answer, but must also do so within an adequately bounded amount of time. For *reactive* real-time systems, that is, systems that typically never terminate, functional correctness is defined as the system exuding correct dynamic behavior in its continuous interaction with the environment. This correctness criteria can be further refined to mean that certain states of the system should be reachable while other states should not be reachable. Moreover, for reactive real-time systems, timing correctness can be defined as a reachable state of the system being reachable within an adequately bounded amount of time.

The use of models for real-time system engineering presents an interesting challenge because to adequately reflect system behavior, models need to describe time-dependent system behavior. Furthermore, for real-time systems, this behavior must be described using quantitative time. When a model is derived from an implemented system, static analysis techniques or measurement techniques can be used to obtain exact measures of software execution times [13]. However, in model-driven engineering approaches [21], the use of models typically happens before the system is implemented. In this situation, estimating execution time of software is an approximate process at best. In this paper, we review the nature of time in real-time systems and how time is represented in popular modeling languages. We also give possible answers on how to address the inherent paradox of modeling time-dependent system behavior in the absence of an implementation. We present the Timed Abstract State Machine (TASM) [26] as a language that contains features to address this paradox.

This paper is divided into five sections in addition to this Introduction. Section 2 reviews the nature of time in real-time systems. Section 3 reviews some basic time concepts and some popular languages that incorporate time as a modeling concept. Section 4 provides possible answers to the inherent paradox of incorporating time in high level models. Section 5 describes how the Timed Abstract State Machine (TASM) language addresses some of these issues. Finally, the Conclusion and Future Work section, Section 6, summarizes the contributions of the paper and explains the additions that are to come in future development.

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

2 The Nature of Time in Real-Time Systems

The nature of time has been a long standing question in philosophy, physics, and mathematics [14]. The *realist* view, pioneered by Isaac Newton [20], considers time as an intrinsic dimension of the universe where events happen in sequence. The contrasting view, pioneered by Leibniz and Kant [16], views time as a purely intellectual framework used to measure and order events. While this paper will not attempt to answer the long standing question of the true nature of time, the philosophical views are helpful to qualify the nature of time in real-time systems. In practice, time appears in real-time systems under both facets. However, in real-time systems, timing correctness does not refer only to the ordering of events, called *qualitative time*, but also refers to the numerical duration between events, called *quantitative time*.

Quantitative time appears in real-time system problems either explicitly or implicitly. Examples of where quantitative time appears explicitly include requirements, the physics of the problem, and constraints of the components of the system. Examples of explicit instances of quantitative time are shown in Table 2. Examples where time appears implicitly, as a side-effect, include software execution time and hardware execution time. Listing 1 shows a brief example of software code, written in the Timeliner scripting language [10]. The code represents a sequence used to maintain cabin temperature between 20 and 25 Celsius degrees. How long does this snippet of Timeliner code take to execute? To answer this question, many other questions need to be answered such as what is the temperature? What are the semantics of this language, e.g., what statements are blocking? How long do the statements block for? What compiler is used to translate the code? What are the details of the hardware platform where the code is executed? Once the code has been written and the system is implemented, these questions can typically be answered to a satisfactory degree of confidence [11].

Listing 1 Sequence TEMP_MONITOR [30]

```
SEQUENCE TEMP_MONITOR
  EVERY 1
    IF TEMPERATURE >= 26 THEN
      SET TRYING_TO_COOL_SYSTEM TO TRUE
      COMMAND COOLING, NEW_STATE=>ON
      WHEN TEMPERATURE <= 22
        SET TRYING_TO_COOL_SYSTEM TO FALSE
        COMMAND COOLING, NEW_STATE=>OFF
      END WHEN
    END IF
    IF TEMPERATURE <= 19 THEN
      COMMAND HEATING, NEW_STATE=>ON
      WHEN TEMPERATURE >= 22
        COMMAND HEATING, NEW_STATE=>OFF
      END WHEN
    END IF
  END EVERY
CLOSE SEQUENCE
```

Source	Example
Requirements	The data in the operator console shall be refreshed 10 times per second
Physics	It takes approximately 5 seconds for a projectile shot straight up in the air at a velocity of 50 m/s to come to rest at its apogee
Components	Pressure sensors can put data on the system bus at a rate of 10Hz

Table 1. Examples of Sources of Explicit Quantitative Time

Even for software, time can also appear implicitly and explicitly. For example, the code in Listing 1 contains one explicit timing statement, the "EVERY 1" statement. This statement tells the runtime system that the sequence shall execute at most once per second. Other examples of explicit timing statements include the statements *sleep* and *wait*, which are present in many programming languages such as C and Java. In real-time system engineering, the explicit sources of quantitative time, outside of software, define the timing constraints of the system that will be built. The goal of real-time system engineering is to build a system that meets these constraints. This is where modeling can add value, by providing increased confidence, before it is physically built, that a system will meet the necessary constraints.

3 Time Concepts in Modeling Languages

In the previous section, we have described how time affects the design and execution of real-time systems. In particular, we have shown an example of software code exuding explicit and implicit timing behavior. In modeling languages, quantitative timing concepts are almost always explicit. The type of modeling described in this paper is *behavioral* modeling, to capture the dynamic aspects of the system. Behavioral modeling is in contrast to *structural* modeling, which captures the static aspects of the system, e.g., a class inheritance hierarchy or a multiplicity relationship. In behavioral modeling, system dynamics are typically represented as some form of transition system where the system transitions from one state to another state based on a set of conditions. Traditional languages to represent state transition systems include finite state automata [29] and Statecharts [15]. For most modeling languages, untimed versions of the language exist and time was added as an extension of the language. This is the case for timed automata [2], time/timed petri nets [7], timed process al-

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

gebra [18], Timed Abstract State Machines (TASM) [26], and the real-time profile of the Unified Modeling Language (UML) [22].

While all of these languages have similarities, they also have significant differences in how they represent and handle time. The two main time models are discrete time and continuous or dense time. In a discrete time model, time progresses in fixed constant steps $dt \in \mathbb{N}^+$. In a continuous time model, time evolves continuously, and any time-related value is taken from the Reals domain ($t \in \mathbb{R}$). Languages also differ on how time evolves. Time can evolve either in states or in transitions. For example, time annotations can be added to petri nets in places or in transitions or in both [7]. The difference lies in whether what we wish to describe is the duration of an action or whether we wish to describe the passage of time. An example of a light switch, modeled in the timed automata of UPPAAL [17] is shown in Figure 1. The model describes the behavior of a lamp [3]. If the lamp is off and the switch is pressed, the lamp will turn on to the low setting. If, after the light has been turned on, the switch is pressed again within 5 time units, the lamp increases its intensity to the bright setting. On the other hand, if the lamp is on and the switch is pressed again, but more than five time units have elapsed, the lamp turns off. This example illustrates a model that describes the passage of time between events. In this model, events are instantaneous but the precise timing between events is of utmost importance.

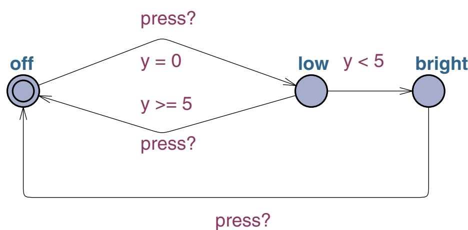

Figure 1. Timed Automaton Describing the Behavior of a Lamp [3]

Another way to represent time is to model events or actions as being durative instead of instantaneous. In the TASM language [24], time is attached to transitions to simulate durative actions. Listing 2 shows the actions of the robot of the production cell system [19], modeled in the TASM language [25]. In the production cell problem, a robot takes commands from a controller and executes these commands. When the robot is instructed to pick up a block, the action takes a certain amount of time to complete until the robot is available again to process other commands. In Listing 2, the action to pick up a block lasts 1 time unit.

Whether a language predominantly favors time passage

Listing 2 Partial TASM Model of a Robot Action to Pick up a Block

```
R1: Arm B at press, block is available -> pick up block
{
  t     := 1;
  power := 2000;

  if armbpos = atpress and armb = empty
     and press_block = available then
    press_block := notavailable;
    press       := empty;
    armb        := loaded;
}
```

or duration of actions in its notation is irrelevant from an expressivity perspective since both types of notations can be used to represent both concepts [4]. The differences lie in what paradigm better fits the problem being addressed. For the specification of real-time systems, and for the modeling of software in general, the term *execution time* is used in numerous contexts. This term refers to the time to execute actions or, in other words, to the duration of actions. Verifying the correctness of a real-time system involves establishing that the durations of the actions of the system meet the time constraints of the requirements and of the problem domain. Listing 3 shows a TASM model of the TEMP_MONITOR sequence from Listing 1. The sequence has been augmented with timing information obtained from [11].

Listing 3 Partial TASM Model of the TEMP_MONITOR Sequence of Listing 1

```
R2: b1 -> b2
{
  t := 2285;

  if temp_seq_b = b1 and temperature >= 26 then
    temp_seq_b               := b2;
    trying_to_cool_system    := True;
    cooling                  := turn_on_device();
}

R3: b1 -> b3
{
  t := 1730;

  if temp_seq_b = b1 and temperature < 26 then
    temp_seq_b := b3;
}

R4: b2 -> b2
{
  t := 1625;

  if temp_seq_b = b2 and temperature > 22 then
    temp_seq_b := b2;
    temp_seq_s := done;
}
```

The TASM model of Listing 2 models time, as enforced by the physics of the problem, namely, the plant dynamics of the robot. The time-dependent behavior of physical properties of the system, such as the one modeled in Listing 2, time can generally be modeled accurately. The TASM

IEEE COMPUTER SOCIETY

model of Listing 3 models the software sequence of Listing 1. This model shows that it is possible to model software and time in modeling languages. However, time was added in the model quite easily because the execution platform is known [11], the precise timing of statements is known [11], and the details of the program are known [30]. In other words, the modeling was performed *after* the system was built. Building models this way is still useful because it enables the analysis of software behavior that cannot typically be performed on the software directly [28]. However, modern model-driven approaches are aimed at complete lifecycle modeling and analysis, typically performed before the system is built. Building system and software models with timing information brings up an interesting question: can we build accurate models of software, including quantitative timing behavior, before the system is implemented?

4 The Time Paradox: Incorporating Time in High-Level Models

The previous two sections explained how time is treated in real-time systems and how modeling languages express time. In this section, we further expand on the paradox encountered when attempting to model system behavior that is closely tied to implementation details. In scheduling theory [8], the task graph [1] is the prevalent modeling pattern. A task graph is a directed graph where nodes represent tasks and edges represent precedence constraints between tasks. Each task is assigned an execution time, that is, a duration. A sample task graph with 7 tasks is shown in Figure 2. The *scheduling problem* is concerned with the time optimal solution to scheduling the set of tasks on *n* processors, while enforcing the precedence constraints.

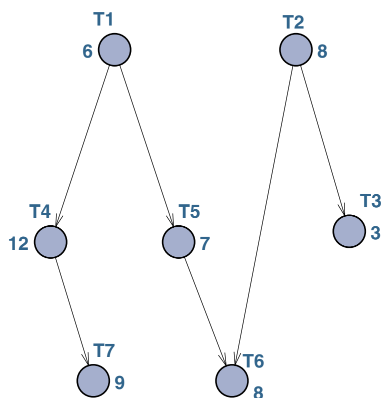

Figure 2. Sample Task Graph

Analogously, the *co-synthesis problem* concerns itself with optimal allocation of a task graph to *processing elements* (e.g., reusable hardware (FPGA), application specific integrated circuits (ASIC), and software) [9]. The similarities between these two problems lie in the existence of a task graph, with known execution times for individual tasks. For the co-synthesis problem, this assumption seems misleading because the execution times will vary depending on which processing element a task is allocated to. On the other hand, for the scheduling problem, the task graph can be derived from an implementation. However, in real-time system engineering, the task graph is an abstraction of an implementation and, conceptually, should be defined before implementation begins. Defining the set of tasks and the dependencies between tasks should be a design decision, not an implementation one. If we rely on the set of tasks to naturally emerge during coding, development will remain an ad-hoc process at best, with little support for predictability. Furthermore, the scheduling problem also assumes that tasks have already been assigned to software, and therefore makes co-synthesis challenging. It is one of the goals of model-driven engineering to remedy ad-hoc system development by structuring engineering activities through the use of models. For real-time systems, how can we build realistic models, such as task graphs, before implementing the system?

The are many possible answers to this paradox. Conceptually, design is and has always been an uncertain process where predictions that may or may not come true are made [6]. Nevertheless, design has proved to be a valuable activity in terms of cost and time saving, even in the face of uncertainty [5]. In a model-driven approach to development, it is highly unlikely that model-driven engineering will be a purely downstream activity flowing one-way from model to implementation. It is more likely that feedback from downstream activities will be incorporated into upstream activities, leading to an iterative model-driven approach, where models are being adjusted as implementation is being developed. Like any other topic in system engineering, experience with building models and experience with engineering using models will dictate the successful use of models in real-time system engineering. Moreover, rooting development around mature and predictable components, as is often the case in aerospace systems, greatly enhances the predictions that can be made by models.

At the modeling level, modeling notations are able to capture the uncertainty involved with annotating models with time. The use of interval semantics for durations gives a lower bound and an upper bound on durations. An example of a TASM specification with duration specified using interval semantics is shown in Listing 4.

Furthermore, the granularity of the model determines whether software times should be included in the model. For system models such as the production cell system [25], the physics of the problem and the time constraints on the system are on a scale much larger (on the order of seconds)

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

Listing 4 TASM Model (partial) of an electronic throttle controller [23] (partial)

```
R1: Driving Mode
{
  t := [2, 5];

  if controller_mode = driving then
    throttle_v := Driving_Throttle_V();
}

R2: Limiting Mode
{
  t := [3, 8];

  if controller_mode = limiting then
    throttle_v := Limiting_Throttle_V();
}
```

than the time scale of the software (on the order of microseconds). Consequently, as it often happens in high-level models, the software is fast enough given the problem definition and time does not need to be included for software in the models. This is certainly the case in the production cell system where controller actions are approximated to be instantaneous.

A model-driven approach should have a notion of *refinement*, that is, a methodology to build models at different levels of abstraction, by gradually adding details to high-level models. Furthermore, the refinement approach should have facilities to show a correspondence between two models at different levels of abstraction. If such a notion is present, time estimates from high level models become constraints on lower level models and, eventually, constraints on implementation. If an implementation cannot satisfy those constraints, the models will need to be adjusted in order to accommodate implementation characteristics. In this view, task graphs can be designed and approximated using high level models, making the scheduling problem and the co-synthesis problem relevant. During the design phase, analyzing schedulability and possible allocations to hardware and software can be useful to drive the implementation.

5 The TASM Approach to Modeling and Analyzing Real-Time Systems

Listings 2, 3, and 4 show partial examples of models built in the TASM language, a language used to model real-time systems. The TASM language is a specification language that incorporates facilities to model and reason about functional behavior and non-functional behavior. The non-functional modeling concepts of the TASM language include time and resource consumption. The language also contains facilities to specify hierarchical composition and parallel composition of system components. The TASM language is the basis for a framework used to engineer real-time systems [24]. The language is implemented into an associated suite of tools, the TASM toolset, to simulate, validate, and formally verify properties of TASM models [12]. The types of properties that can be verified in the toolset include completeness and consistency [27], and execution time characteristics [28].

The TASM language uses durative actions as the underlying time model. This model is adequate to describe real-time system behavior where duration of actions and temporal dependencies are the central concern. The time model is combined with an additive resource consumption model to capture parallel behavior. The TASM language has been used to model and analyze an electronic throttle controller [23], the production cell system [25], and the Timeliner system [28], a scripting environment currently in use on the International Space Station. The TASM language and toolset seek to address some of the paradoxes exposed in earlier sections of this paper. The interval semantics of durative actions help mitigate the uncertainty of estimating time. Furthermore, a theory of refinement is currently being developed for the language. The end goal of the language and framework is to provide end-to-end traceability from high-level models, all the way down to implementation. This will be achieved through the refinement theory and through code generation techniques.

6 Conclusion and Future Work

In this paper, we have presented some of the important issues typically encountered when incorporating time in the modeling of hardware and software systems. This paper also raises questions about the feasibility of model-driven engineering for real-time systems. We have reviewed basic time concepts in popular modeling languages and some of the paradoxes that naturally occur in the engineering of real-time systems using models. We have partially answered some of these questions, and we have presented the Timed Abstract State Machine (TASM) approach to modeling real-time systems as a potential solution.

In future work, we plan to model and analyze more examples of real-time systems using the TASM language and toolset. As we develop our refinement theory, we hope to achieve end-to-end traceability of lifecycle activities, from high-level models all the way down to implementation.

References

[1] Y. Abdeddaïm, A. Kerbaa, and O. Maler. Task Graph Scheduling using Timed Automata. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[2] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2), 1994.

29th International Conference on Software Engineering Workshops(ICSEW'07)
0-7695-2830-9/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

[3] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, volume 3185 of *LNCS*. Springer-Verlag, 2004.

[4] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petruccci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification*. Springer-Verlag, 2001.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[6] D. Budgen. *Software Design*. Addison Wesley, 2nd edition, 2003.

[7] A. Cerone and A. Maggiolo-Schettini. Time-based Expressivity of Time Petri Nets for System Specification. In *Theoretical Computer Science*, volume 216. Springer-Verlag, 1999.

[8] A. K. Cheng. *Real-Time Systems: Schedulability, Analysis, and Verification*. John Wiley and Sons, 2003.

[9] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-Software Co-Synthesis of Embedded Systems, booktitle = Design Automation Conference, pages = 703–708, year = 1997.

[10] Draper Laboratory. The Timeliner User Interface Language (UIL) System for the International Space Station. Technical Report available from http://timeliner.draper.com.

[11] Draper Laboratory. User Interface Language CPU Performance Report for the ISS Timeliner. Technical Report CSDL 306647, September 1997. Available from http://timeliner.draper.com.

[12] Embedded Systems Laboratory. The Timed Abstract State Machine Language and Toolset. http://esl.mit.edu/tasm.

[13] J. Engblom, A. Ermedahl, M. Nolin, J. Gustafsson, and H. Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *International Journal on Software Tools for Technology Transfer*, 4:437–455, October 2003.

[14] P. Galison. *Einstein's Clocks, Poincare's Maps: Empires of Time*. W. W. Norton and Company, 2004.

[15] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[16] I. Kant. *Critique of Pure Reason*. Dover Publications, 2003.

[17] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[18] L. Leonard and G. Leduc. A Formal Definition of Time in LOTOS. *Formal Aspects of Computing*, 10, 1998.

[19] C. Lewerentz and T. Lindner. Production Cell: A Comparative Study in Formal Specification and Verification. In *KORSO - Methods, Languages, and Tools for the Construction of Correct Software*, 1995.

[20] I. Newton. *The Principia*. University of California Press, 3rd edition edition, 1999. Translated by I. Bernard Cohen and Anne Whitman.

[21] Object Management Group, Inc. MDA Guide Version 1.0.1. OMG Specification, June 2003.

[22] Object Management Group, Inc. UML Profile for Schedulability, Performance, and Time Specification. OMG Specification, January 2005.

[23] M. Ouimet, G. Berteau, and K. Lundqvist. Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. In *Proceedings of the Satellite Events of the 2006 MoDELS Conference*, LNCS, October 2006.

[24] M. Ouimet and K. Lundqvist. The Hi-Five Framework and the Timed Abstract State Machine Language. In *Proceedings of the The 27th IEEE Real-Time Systems Symposium - Work in Progress Session*, December 2006.

[25] M. Ouimet and K. Lundqvist. The Production Cell: Model and Analysis in the TASM Language, December 2006. Technical Report ESL-TIK-000209, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[26] M. Ouimet and K. Lundqvist. Timed Abstract State Machines: An Executable Specification Language for Reactive Real-Time Systems, May 2006. Technical Report ESL-TIK-000193, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[27] M. Ouimet and K. Lundqvist. Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. In *Proceedings of the 3rd International Workshop on Model-Based Testing (MBT '07), Satellite Workshop of ETAPS '07*, April 2007.

[28] M. Ouimet and K. Lundqvist. Verifying Execution Time using the TASM Toolset and UPPAAL, January 2007. Technical Report ESL-TIK-000212, Embedded Systems Laboratory, Massachusetts Institute of Technology.

[29] M. Sipser. *Introduction to the Theory of Computation*. Springer-Verlag, 2003.

[30] S. M. Stern. An Extensible Object-Oriented Executor for the Timeliner User Interface Language. Master's thesis, Massachusetts Institute of Technology, 2005.

0-7695-2830-9/07 $20.00

© 2007 IEEE

IEEE COMPUTER SOCIETY