Towards a More Flexible Timing Definition Language

Tomasz Kloda Bruno d'Ausbourg Luca Santinelli
ONERA
Toulouse, France
name.surname@onera.fr

Time-triggered languages permit to model real-time system temporal behavior by assigning system activities to the particular time instants. At these precise instants system observes the controlled object and, depending on the analysis of its state, invokes the appropriate actions. This fine-grained control of system temporal evolution enables value and time deterministic programming. Up-to-date time-triggered frameworks allow to model multi-modal and multi-modular real-time systems. However, their timing verification imposes some constraints on the computational task model and system reactivity. By adapting scheduling analysis techniques based on the processor demand instead of processor utilization factor, these limitations can be overcome and a more flexible framework may be proposed.

1 Introduction

The development of embedded software is a highly platform dependent process. The main difficulty lies in both formulating the functional specification of the system and correctly determining its temporal behavior. While the former is facilitated by high level programming languages which abstract from many hardware aspects, getting the expected temporal characteristic of the system involves usually much more efforts due to the implementation of scheduling policies, synchronization and inter-processes communication protocols.

To answer the problem of managing efficiently these two crucial for the correctness of the system aspects, time-triggered languages were devised. These languages clearly separate the functional part of applications from their timing definition. Applications are specified through two descriptions: their time definition, expressed in a time-triggered language, and the functional code of tasks, expressed in any programming language (e.g. C). Functional code does not attempt to control time as the role of specifying this system's property is shifted completely towards time definition part which defines exact points in time at which communication and processing activities are initiated. The state of the system and its operational modes, can change only at these well-defined beforehand instants. A dedicated compiler generates, based on both descriptions, an execution control code that will have to be interpreted by a time-triggered execution system built on top of a selected target platform.

Clearly separating timing definition and functionality specification makes the application development process less platform oriented and allows designers to focus more on the abstract system control aspects than on the concrete execution mechanisms such as task scheduling or time management. The programming of these mechanisms is devoted to the compiler.

Logical Execution Time Time-triggered languages provide the designer with the powerful programming abstraction of *Logical Execution Time* (*LET*) [6, 7]. The application designer assigns a *LET* to each task. The *LET* semantics fixes a time interval as an abstract and logical view of the task execution. The task is considered to start logically at the start instant of the *LET* interval. Its input data are read exactly at this start time. The task is considered to complete logically at the end

Submitted to:
QAPL 2014

instant of the *LET* interval. Its output data (the results of its computation) are produced exactly at this end instant even if the computation was physically terminated before. Such assumption leads to a well-defined and deterministic interaction between different parallel activities: it is always known at which time a value will be produced and then which value is in use at a given time. Therefore the observable behavior of the system is exactly the logical one and is independent from its physical execution as depicted on Figure 1.

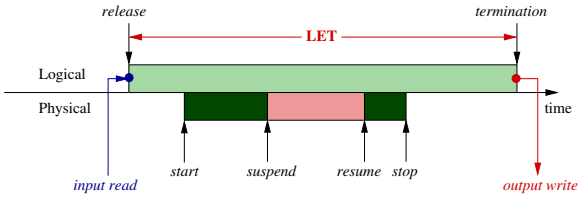

Figure 1: Logical Execution Time.

Timing Definition Languages The concept of LET was introduced for the first time within the *Giotto* [1, 6] programming language. Giotto assigns a *LET* to each *task* that executes some piece of code. Subsequent task invocations are separated by a period that is equal to its LET. Different tasks communicate between them as well as with sensors and actuators, which exchange data with environment, by means of *ports* only. Reading of input ports and sensors take place exactly at the beginning of task LET interval while writing to output ports and actuators at its end. Several concurrent tasks may be grouped into a *mode* that is invoked when the environment or system is in some specific state that should by handled by this particular mode. Giotto program can be only in one mode at a time. Tasks are invoked within the mode, which is characterized by some period, at the declared frequencies and can be removed or added when switching from one mode to another. Giotto was further extended by *xGiotto* [4] where tasks can be released also at the occurrence of the external events. *Timing Definition Language* (*TDL*) [3, 10, 12], another successor of Giotto, introduces decomposition of large systems into executing concurrently *modules* (components). Each module runs in one mode at a time and can switch to another independently from other modules. *Hierarchical Timing Language* (*HTL*) [5] makes use of *abstract tasks*. These abstract tasks may be refined by groups of concrete tasks and each refining group of concrete tasks must behave in full accordance with the timing constraints of the abstract task it refines.

2 An Extended Timing Definition Language

Each of the above cited frameworks can be used successfully and introduces new features for the concepts laid down in Giotto. But all of them rely on the same task model structure. In this paper, we propose a new time-triggered framework, named *Extended Timing Definition Language* (*E-TDL*), that tries to enhance the basic task model used in TDL while keeping compositional and multi-modal structure brought by this language. Furthermore, introducing such a new task model permits to devise also new mode switching mechanisms.

The E-TDL Task Model Classically, a time-triggered task $\tau_i = (C_i, LET_i)$ is defined by its worst case execution time C_i and the Logical Execution Time LET_i that is equal to its period T_i ($T_i \equiv LET_i$). The input ports of τ_i are read at the beginning of LET_i interval, τ_i executes for at most C_i time within LET_i and its computation results are written on the output ports exactly at the end instant of LET_i. A new instance of τ_i is then released immediately to repeat this cycle. In this model, task periods and *LETs* are always the same even though they denote two different notions. The task period should denote the time intervals that separate subsequent task invocations. The task *LET* denotes its logical and abstract computation time. By distinguishing these two notions, it would be possible to shorten the response time of a task without unnecessarily increasing the frequency of its executions. Moreover, the start time of a task could

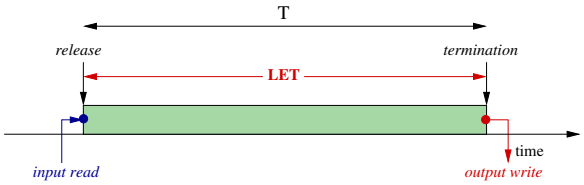

Figure 2: Giotto/TDL task model.

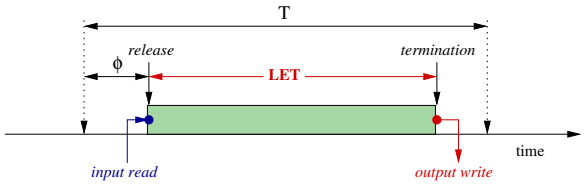

Figure 3: E-TDL task model.

be moved forward allowing a later read of inputs and thus giving more freedom in the modeling of the inter-task data flows. Therefore, we suggest to characterize an E-TDL task $\tau_i = (\Phi_i, C_i, LET_i, T_i)$ with four parameters: an offset Φ_i, a worst case execution time C_i, a Logical Execution Time LET_i and a period T_i. An E-TDL $n-th$ task instance ($n \in \mathbb{N}$) is invoked at time $t_r = \Phi_i + nT_i$ when its input ports are read. It may run for at most C_i time units in the LET_i interval till the time instant $t_d = \Phi_i + nT_i + LET_i$ when its results are written onto its output ports. Figures 2 and 3 show the difference between the both task models.

E-TDL Mode Switch When the system detects a condition change in its environment, it can follow this evolution by switching from its current operating mode to a new mode. Some tasks are deleted from the currently executed task set, others are added and still others modify their parameters or remain unaffected. Mode switching mechanisms are a bit different in Giotto [6] and in TDL. In both cases, the well timedness condition ensures that mode switches do not terminate logical execution of any task. In TDL mode switches may occur only at time instants when all tasks are logically completed. In Giotto, if a mode switch occurs when a task is logically running, then the same task must be present, with the same timing properties, in the new mode. So, a mode switch jumps to the time point in the new mode where the logical execution of unchanged tasks is exactly in the same state it was in the previous mode.

E-TDL follows the same mode switching rule as in TDL: mode switches occur only when all tasks are logically completed. Because periods and *LETs* are the same in TDL, mode switches may occur only at time instants corresponding to hyperperiods of tasks in the current mode. The E-TDL task model allows tasks to be logically completed before the end of their periods. This fact may generate additional intervals where all tasks are logically completed. Mode switches may occur during these intervals without breaking the well timedness rule. The allowable mode switch points, in the two E-TDL tasks mode of Figure 4, are depicted by the blue intervals with inclined lines. Moreover, by aborting execution of some group of tasks, whose outputs, in new conditions, are no longer vital for the consistency of the system, the number of valid switching points may be increased. In [9] Martinek et al. applied such mode switch mechanism together with a sporadic task model ($\tau_i = (C_i, LET_i, T_i)$) to the Giotto. E-TDL aims to integrate the above mentioned propositions in the compositional framework, containing multiple parallel applications, while in Giotto only one program is executed at a time.

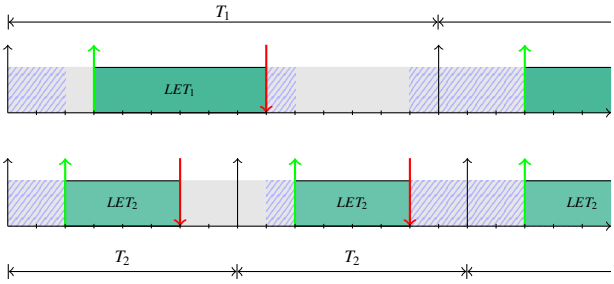

Figure 4: E-TDL mode switch instants.

3 Schedulability Analysis

To ensure that a system modeled with E-TDL respects all its timing constraints, processing time demanded by tasks should be quantified and compared to the available resources. In a first stage of our

study, we consider *Earliest Deadline First* (*EDF*) [8] scheduling algorithm. The results obtained for TDL [3] are based on the *processor utilization* criterion [8] because the LET of tasks is equal to their period. These results cannot be generalized for E-TDL. So, the *processor demand* criterion [2] should be applied in this case. The processor demand criterion states that a task set is feasible if the cumulative demand of the computation made by tasks in any interval is never larger than this interval length. *Real Time Calculus* [11] proposes a method that supports any task activation pattern, not only a purely periodic one with uni-modal behavior, and performs compositional analysis of the system where different modules execute concurrently. The maximal demand that can be generated by each individual module over any interval of given length is computed. Then, the demands from different modules are summed up and compared to the minimal available resources for this interval length. If it occurs that the timing definition specifies that the summed up intervals cannot be observed simultaneously, the result of the test can be overestimated. We take this observation as the starting point of our future work. We would like to reduce this pessimism by considering only the global schedules that are made up of task activation patterns from the distinct modules that start at the same time instant.

References

[1] *Giotto*. Available at `http://embedded.eecs.berkeley.edu/giotto/`.

[2] Sanjoy K. Baruah, Rodney R. Howell & Louis Rosier (1990): *Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor*. Real-Time Systems 2, pp. 301–324.

[3] Emilia Farcas (2006): *Scheduling Multi-Mode Real-Time Distributed Components.* Ph.D. thesis, Department of Computer Science, University of Salzburg.

[4] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch & Marco A. A. Sanvido (2004): *Event-Driven Programming with Logical Execution Times*. In: Hybrid Systems Computation and Control, Lecture Notes in Computer Science 2993, Springer, pp. 357–371.

[5] Arkadeb Ghosal, Alberto L. Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger & Daniel T. Iercan (2006): *A Hierarchical Coordination Language for Interacting Real-Time Tasks.* In: Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT), pp. 132–141.

[6] Thomas A. Henzinger, Benjamin Horowitz & Christoph M. Kirsch (2000): *Giotto: A Time-triggered Language for Embedded Programming*. In: Proceedings of the IEEE, Springer-Verlag, pp. 166–184.

[7] Christoph M. Kirsch & Ana Sokolova (2012): *The Logical Execution Time Paradigm*. In: Advances in Real-Time Systems, pp. 103–120.

[8] C. L. Liu & James W. Layland (1973): *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the Association for Computing Machinery 20(1), pp. 46–61.

[9] Norbert Felix Martinek & Werner Pohlmann: *Mode Switching in GIA An Ada based Real-Time Framework*. Department of Scientific Computing, University of Salzburg, Austria.

[10] Wolfgang Pree, Josef Templ, Peter Hintenaus, Andreas Naderlinger & Johannes Pletzer (2011): *TDL - Steps Beyond Giotto: A Case for Automated Software Construction.* Int. J. Software and Informatics 5(1-2), pp. 335–354.

[11] Nikolay Stoimenov, Simon Perathoner & Lothar Thiele (2009): *Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling*. In: Proceedings of Design, Automation and Test in Europe, 2009 (DATE 09), IEEE, Nice, France, pp. 99–104.

[12] Josef Templ (2008): *Timing Definition Language (TDL) Specification 1.5*. Technical Report T024, Department of Computer Science, University of Salzburg, Austria. Available at `http://www.uni-salzburg.at/fileadmin/multimedia/SRC/docs/publications/T024.pdf`.