Elektrotechnik & Informationstechnik (2007) 124/1/2: 9–12. DOI 10.1007/s00502-006-0410-7

Portable and composable real-time software – a disruptive approach

W. Pree

The reliability and quality of embedded systems suffer from the fact that the state-of-the-art in embedded software development is significantly behind modern programming in non-real-time domains. A major difference is that the platform, consisting in particular of processors, communication architecture, topology of a distributed system, communication protocols and the operating system(s), needs to be defined first. The software is then adjusted to that platform. Changes of the platform typically imply a major adaptation if not a development from scratch. Our research focuses on the envisioned paradigm shift that allows the definition of the timing and functionality behavior independent from a specific platform. Automatic code generation ensures that the executable code on a specific platform behaves exactly as specified. Thus the platform and the mapping to it are defined at the end of the development process. This disruptive approach allows, as typical for disruptive technologies, a cost-quality relation that was previously regarded as impossible: significantly better quality, reliability and portability at a fraction of today's development costs.

Keywords: real-time components; model-based development; dependability; portability

Ein Paradigmenwechsel hin zu portablen Echtzeit-Softwarekomponenten.

Zuverlässigkeit und Qualität von eingebetteten Systemen leiden darunter, dass die Softwareentwicklung für diese Domäne, verglichen mit dem Stand der Technik in anderen Anwendungsbereichen, zu wünschen übrig lässt. Ein wesentlicher Unterschied besteht darin, dass die Software auf eine bestimmte Plattform zugeschnitten wird. Unter Plattform verstehen wir die Prozessoren, die Kommunikationsarchitektur, die Topologie eines verteilten Systems, das Betriebssystem sowie die verwendeten Protokolle. Änderungen der Plattform führen meist zu substantiellen Änderungen oder sogar zu Neuentwicklungen. Forschungsarbeiten konzentrieren sich darauf, einen Paradigmenwechsel herbeizuführen, so dass das Zeitverhalten und die Funktionalität unabhängig von der Plattform definiert werden können. Der Plattform-spezifische Code wird durch automatische Code-Generatoren erzeugt. Mit diesem Paradigmenwechsel ist es möglich, eine Kosten-Nutzen-Relation zu erhalten, die bisher nicht realistisch erschienen ist: signifikant bessere Qualität, Zuverlässigkeit und Portierbarkeit zu einem Bruchteil der heutigen Entwicklungskosten.

Schlüsselwörter: Echtzeit-Komponenten; Modell-basierte Entwicklung; Zuverlässigkeit; Portierbarkeit

Eingegangen am 27. September 2006, angenommen am 23. November 2006
© Springer-Verlag 2007

1. Introduction

Traditional development of software for embedded systems is highly platform specific. The hardware costs are reduced to a minimum whereas high development costs are considered acceptable in case of large quantities of devices being sold. However, with more powerful processors even in the low cost range, we observe a shift of functionality from hardware to software and in general more ambitious requirements. A luxury car, for example, comprises about 80 electronic control units interconnected by multiple buses and driven by more than a million lines of code. In order to cope with the increased complexity of the resulting software, a more platform independent ''high-level'' programming style becomes mandatory. In case of real-time software, this applies not only to functional aspects but also to the temporal behavior of the software. Dealing with time, however, is not covered appropriately by any of the existing component models for high-level languages.

A particularly promising approach towards a high-level component model for real time systems has been laid out in the Giotto project *(http://www-cad.eecs.berkeley.edu/~fresco/giotto) (Henzinger, Horrowitz, Kirsch, 2001a, b; Henzinger et al., 2003)* by introduction of logical execution time (LET), which abstracts from the physical execution time on a particular platform and thereby abstracts from both the underlying execution platform and the communication topology. Thus, it becomes possible to change the underlying platform and even to distribute components between different nodes without affecting the overall system behavior. Giotto, however, is primarily an abstract mathematical concept and there exist only simple prototype implementations, which show some of the potential of LET.

This paper presents a component model, named TDL (Timing Definition Language) *(Templ, 2004),* that has been developed in the course of the MoDECS[1] project at the University of Salzburg, as a successor of Giotto. It shares with Giotto the basic idea of LET but introduces additional high-level concepts for structuring large real time systems.

In the following, we shall start with an explanation of LET and proceed with an overview of the TDL component model. An outlook of envisioned future TDL extensions rounds out the paper.

2. Logical execution time (LET)

LET means that the observable temporal behavior of a task is independent from its physical execution *(Henzinger, Horrowitz, Kirsch, 2001a).* It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. Figure 1 shows the relation between logical and physical task execution.

[1] The MoDECS project (www.MoDECS.cc, 2003–2005) was supported by the FIT-IT Embedded Systems grant 807144 (www.fit-it.at).

Pree, Wolfgang, O. Univ.-Prof. Dipl.-Ing. Dr., C. Doppler Lab Embedded Software Systems, University of Salzburg, Jakob-Haringer-Straße 2, 5020 Salzburg, Austria (E-mail: pree@SoftwareResearch.net)

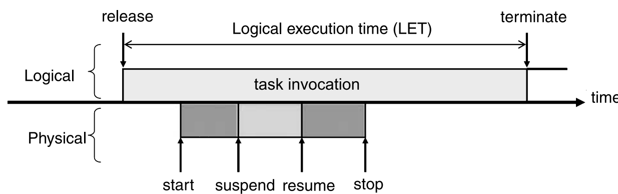

Fig. 1. Logical execution time

The inputs of a task are read at the release event and the newly calculated outputs are available at the terminate event. Between these, the outputs have the value of the previous execution.

LET introduces a delay for observable outputs, which might be considered a disadvantage. On the other hand, however, LET provides the cornerstone to deterministic behavior, platform abstraction and well-defined interaction semantics between parallel activities *(Kirsch, 2002)*. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. As we will see later, LET also provides the foundation for transparent distribution. A recent extension of TDL has substantially improved the support for digital controllers and preserved the semantics of existing TDL features.

3. TDL component model

Based on the concept of LET, Giotto introduces the notion of a *mode* as a set of periodically executed activities. The activities are task invocations (according to LET semantics), actuator updates, or mode switches. All activities can have their own rate of execution and all activities can be executed conditionally. Actuator updates and mode switches are considered to be much faster than task invocations, thus they are executed in logical zero time. The set of all modes reachable from a distinguished start mode constitutes the Giotto *program*.

Our successor of Giotto, named TDL (Timing Definition Language), extends these concepts by the notion of the *module*, which is a named Giotto program that may import other modules and may export some of its own program entities to other client modules. Every module may provide its own distinguished start mode. Thus, all modules execute in parallel or in other words, a TDL application can be seen as the parallel composition of a set of TDL modules. It is important to note that LET is always preserved, i.e. adding a new module will never affect the observable temporal behavior of other modules. It is the responsibility of internal scheduling mechanisms to guarantee conformance to LET, given that the worst-case execution times (wcet) and the execution rates are known for all tasks. Figure 2 sketches a sample module with two modes containing two cooperating tasks each.

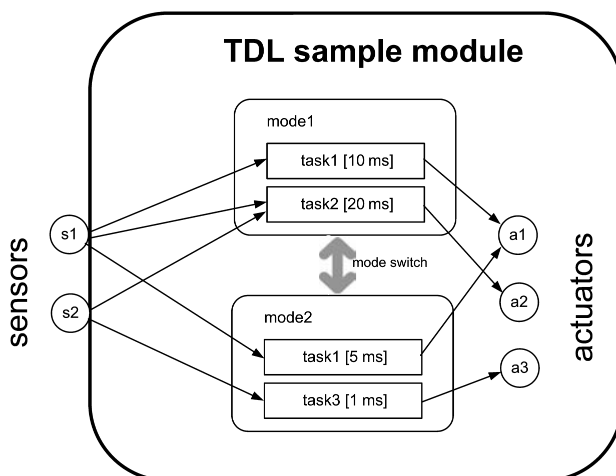

Fig. 2. Visual representation of a TDL module

Parallel tasks within a mode may depend on each other, i.e. the output of one task may be used as the input of another task. All tasks are logically executed in sync and the dataflow semantics is defined by LET.

Modules support an export/import mechanism similar to modern general purpose programming languages such as Java or C#. A service provider module may export a task's outputs, which in turn may be imported by a client module and used as input for the client's computations. All modules are logically executed in sync and again the dataflow semantics is defined by LET. Modules are a top-level structuring concept that serves multiple purposes: (1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, (2) modules provide parallel composition of real time applications, (3) modules serve as units of loading, i.e. a runtime system may support dynamic loading and unloading of modules, and (4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion). The possibility to distribute TDL modules across different computation nodes leads us to the notion of transparent distribution as explained below.

4. Transparent distribution

We define the term *transparent distribution* in the context of hard real-time applications with respect to two points of view. Firstly, at runtime a TDL application behaves exactly the same, no matter if all modules (i.e. components) are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed. Secondly, for the developer of a TDL module, it does not matter where the module itself and any imported modules are executed. The TDL tool chain and runtime system frees the developer from the burden of explicitly specifying the communication requirements of modules. It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application.

The advantage of transparent distribution for a developer is that the TDL modules can be specified without having the execution on a potentially distributed platform in mind. The mapping of modules to computation nodes is defined separately. Nevertheless, the functional and temporal behavior of a system is exactly the same no matter where a component is executed.

The only place where distribution is visible is for the system integrator, who must specify the module-to-node assignment by means of a configuration file. This file, which is based on the available processing nodes with their peripherals (e.g., directly connected sensors and actuators) and network resources, is used as input for the TDL tool chain. Figure 3 shows an example of a set of four TDL modules distributed across three nodes.

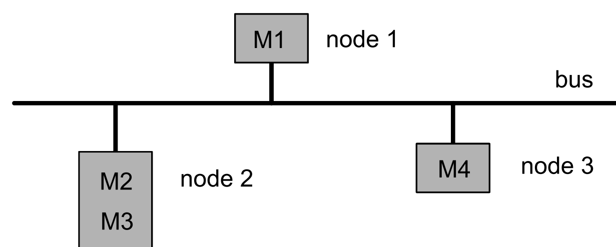

Fig. 3. Example of distributed modules

Figure 3 shows the screenshot of the TDL:VisualDistributor tool that allows the visual and interactive specification of the platform and the module-to-node assignment. In the example we have defined a FlexRay cluster consisting of three nodes (node1, node2, and node3). The modules M1, M2, M3, and M4 are assigned to these

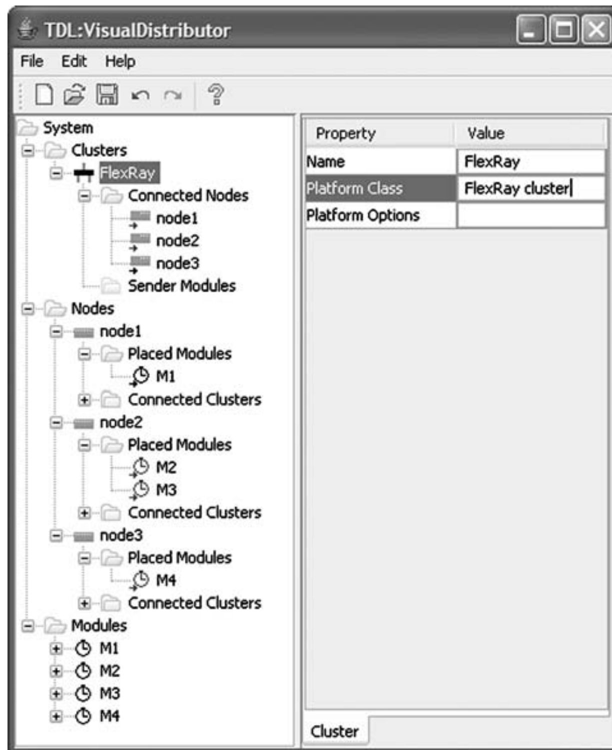

Fig. 4. Mapping TDL modules to nodes

nodes according to Fig. 3. Assigning a module to a node is a straightforward drag-and-drop operation in the TDL:VisualDistributor.

5. TDL tool chain

Figure 5 shows the TDL core tool chain as well as which inputs the tools require and which outputs they produce. The compiler processes TDL source code and generates an abstract syntax tree (AST) representation of the TDL program as intermediate format as well as the so-called embedded code (E-code) (*Henzinger, Kirsch, 2002*), which describes when to release a task. The plug-in architecture of the compiler allows the extension of the tool with any number of tools that rely on the AST.

The bus scheduler is such a plug-in tool that generates the bus schedule, based on a configuration file. The configuration file simply

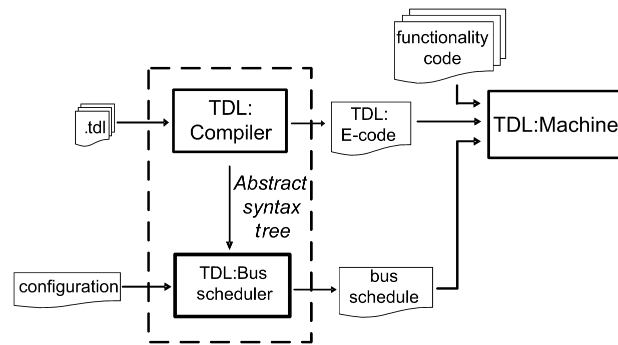

Fig. 5. TDL tool chain core

contains a list of computing nodes that comprise the particular platform, the assignment of TDL modules to computing nodes, and the physical properties of the communication infrastructure. This can be specified interactively with the TDL:VisualDistributor tool.

The runtime environment of TDL is structured in several layers and is based on virtual machines. Tasks are executed according to the LET semantics under the control of the E-machine *(Henzinger, Kirsch, 2002):* a virtual machine that executes E-code instructions. Scheduling decisions can be executed by the OS scheduler or better by the S-machine *(Henzinger, Kirsch, Matic, 2003):* a virtual machine that executes scheduling-code (S-code) instructions.

6. TDL+

The aim of TDL+ is to achieve a comparable leap forward as the Giotto and MoDECS projects have delivered and thus narrow the gap close to zero between software development for non-embedded, non-real-time domains and embedded, hard real-time systems. In other words, we aim at what is called model-based development of embedded control systems after mastering the major initial hurdles in the Giotto and MoDECS projects. Model-based development means that appropriate abstractions narrow the gap between the domain problem and its solution. Table 1 sketches in the right column, in bold face, the gap, that is, the missing steps towards full-fledged model-based development of embedded control systems.

The left column of Table 1 summarizes the history of programming languages: This history is a series of successful inventions of abstractions for general-purpose programming: (1) Basic high-level programming constructs (sequence statements, conditional statements, loop statements) together with hierarchical structuring means called func-

Table 1. Abstractions for general purpose programming (left) versus for embedded programming (right)

Abstractions in general-purpose programming languages	Abstractions in embedded, hard real-time programming languages
(5) Aspect-Oriented Programming (AOP) harnesses meta-programming to improve the modularization of software	(5) TDL-AOP for a better structuring of embedded software and avoidance of replicated code ⇒ well-structured, maintainable code
(4) domain-specific language extensions, for example, class/component libraries for Graphical User Interfaces (GUIs)	(4) relevant aspects of control theory: 10/1 rule for actuator updates, advance calculation ⇒ high-quality controllers with minimal hardware resources
(3) object-orientation (abstract data types, inheritance, dynamic binding)	(3) dynamic loading of TDL modules; instantiable TDL modules (e.g., for specifying redundancy in fault-tolerant systems); TDL mode extension through inheritance ⇒ reusability of TDL components
(2) modules (information hiding through abstract data structures) as means for component-based reuse and for structuring large software systems	(2) TDL module ⇒ transparent distribution
(1) basic language abstractions and hierachical structuring constructs (functions and procedures) as foundation of high-level, platform-independent programming languages	(1) Logical Execution Time (LET) ⇒ determinism, portability

tions and procedures set the stage of significanty more platform-independent programming in the late 1950s and 1960s. This ended the tedious machine level and assembly coding. (2) The module was invented as means to structure software in the large and to provide reusable software components. Modula and Ada as typical representatives of modular languages were defined in the 1970s and 1980s. (3) Object-oriented languages improved the reusability of modules. The class construct as instantiable module was originally provided by Simula in the 1960s. Smalltalk (1970s) and later on C++ (1980s), Java (1990s) and C# (2000) pushed object-orientation as common programming paradigm into the main stream. (4) Domain-specific extensions of languages have been delivered as class or component libraries since the 1980s. GUI (Graphical User Interface) libraries represent one important example of such a domain-specific extension. (5) So-called aspect-oriented programming (AOP), proposed in the 1990s, is heralded as post-object-oriented programming paradigm. It harnesses meta-programming to be able to modify the semantics of a programming language in order to improve the modularization of software.

The items (1) and (2) in the right column of Table 1 summarize the achievements of the Giotto and MoDECS projects towards closing the gap between embedded and non-embedded software development: The pioneering LET abstraction delivers the important software property determinism and together with the TDL component model forms the foundation of transparent distribution (see section 4). TDL+ aims at delivering (3), (4) and (5) in the right column of Table 1. Radical innovation is required to invent and shape these missing abstractions: as (1) and (2) corroborate, the abstractions of high-level general purpose programming languages cannot be transferred directly to the real-time domain. For example, a module in general-purpose programming languages comprises other entities than a TDL module which has been defined according to the characteristics of hard real-time embedded systems. The same is true for the LET abstraction. The concepts and intentions of general-purpose programming language abstractions serve only as inspiration for the corresponding abstractions in the embedded hard real-time domain.

6.1 Expected results

TDL has already been seamlessly integrated with Matlab/Simulink, which has been established as industry standard for modeling control applications. We accomplished the integration in the MoDECS project and called it the TDL:VisualCreator tool. This tool will be extended to comprise the TDL+ extensions.

The fully automated code generation and the TDL run-time system guarantee that the behavior of a TDL module corresponds exactly to its specification. This is a characteristic and a principal advantage of model-based software development. The conventional way of showing the correctness of the code generators is by testing various sample applications. Instead of just testing the code generators we aim at verifying the most critical generator tool in the TDL+ project, the bus schedule generator. This represents another significant contribution to full-fledged model-based software development.

To sum up, the principal goals of the planned future research activities are the following:

- full-fledged model-based development by inventing and implementing abstractions (3), (4), and (5), resulting in the modeling language TDL+; some of the control engineering enhancements (4) have already been accomplished
- corresponding extensions of the MatLab/Simulink integration, that is, the TDL+:VisualCreator tool
- verification of the bus schedule generator

6.2 Practical relevance

The TDL tools are currently used and evaluated by automotive suppliers and OEMs, in particular in the realm of time-triggered distributed systems development with FlexRay and with EtherCAT. TDL will also be used in the realm of the 2007 DARPA Urban Grand Challenge in the vehicle developed by the University of California, Berkeley.

7. Conclusion

The LET abstraction invented in the realm of the Giotto project paved the way for transparent distribution in real-time systems. From our own experience and the feedback from various industry partners we are convinced that this novel approach is a breakthrough that will lead to significantly more robust embedded software and will at the same time reduce the costs of development and integration testing.

Acknowledgements

We thank the MoDECS project team at the University of Salzburg for providing valuable input during informal discussions and group meetings. This research was supported in part by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through its Bundesminsterium für Verkehr, Innovation und Technologie.

References

Giotto Project, http://www-cad.eecs.berkeley.edu/~fresco/giotto/.

Henzinger, T. A., Kirsch, C. M. (2002): The embedded machine: predictable, portable real-time code. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI): 315–326.

Henzinger, T. A., Kirsch, C. M., Matic, S. (2003): Schedule carrying code. In: Springer: Proc. of the 3rd Int. Conf. on Embedded Software (EMSOFT), LNCS.

Henzinger, T. A., Horowitz, B., Kirsch, C. M. (2001a): Giotto: A time-triggered language for embedded programming. Springer: Proc. of the 1st Int. Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211: 166–184.

Henzinger, T. A., Horowitz, B., Kirsch, C. M. (2001b): Embedded control systems development with Giotto. ACM Press: Proc. of the Int. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES): 64–72.

Henzinger, T. A., Kirsch, C. M., Sanvido, M. A. A., Pree, W. (2003): From control models to real-time code using Giotto. IEEE Control Systems Magazine 23 (1): 50–64.

Kirsch, C. M. (2002): Principles of real-time programming. In: Proc. of EMSOFT 2002, Grenoble LNCS, 2491.

Templ, J. (2004): TDL specification and report. Technical Report C059, Department of Computer Science, University of Salzburg, http://www.cs.uni-salzburg.at/pubs/reports/T001.pdf/.

The Author

Wolfgang Pree

is a Professor of Computer Science at the University of Salzburg, Austria. His research focuses on software construction, in particular methods and tools for automating the development of real-time embedded software.