# Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm

Julien Hennig, Hermann v. Hasseln, Hassan Mohammad
Daimler AG
Email: {julien.hennig, hermann.v.hasseln, hassan.mohammad}@daimler.com

Stefan Resmerita, Stefan Lukesch, Andreas Naderlinger
Department of Computer Sciences, University of Salzburg
Email: {stefan.resmerita, stefan.lukesch, andreas.naderlinger}@cs.uni-salzburg.at

***Abstract*—The growing demand for computing power in automotive applications can only be satisfied by embedded multi-core processors. Significant parts of such applications include OEM-owned legacy software, which has been developed for single-core platforms. While the OEM is faced with the issues of parallelizing the software and specifying the requirements to the ECU supplier, the latter has to deal with implementing the required parallelization within the integrated system. The Logical Execution Time (LET) paradigm addresses these concerns in a clear conceptual framework. We present here initial steps for applying the LET model in this respect: (1) Parallelization of legacy embedded control software, by exploiting existing inherent parallelism. The application software remains unchanged, as adaptations are only made to the middleware. (2) Using the LET programming model to ensure that the parallelized software has a correct functional and temporal behavior. The Timing Definition Language (TDL) and associated tools are employed to specify LET-based requirements, and to generate system components that ensure LET behavior. The work describes two conceptual ways for integrating TDL components in AUTOSAR.**

## I. Introduction

By now the multi-core revolution has hit embedded computing full force. Most embedded microcontroller manufacturers offer multi-core based architectures as part of their mid and high-end lineups. As was the case for mainstream computing, porting, transforming and rewriting substantial legacy software to make effective use of the new processors' parallelism is trailing the advent of the new architectures. Parallelizing proven-in-use time-critical embedded control software is posing significant challenges that call for rigorous software design patterns and programming models. For a number of years, we participated in industrial development projects implementing real-time control applications for first-of-their-kind multi-core-based electronic control units (ECUs) in the chassis, driver-assistance and powertrain domains. We saw two distinct activities involved in such implementation efforts. The first activity is to expose parallelism in the legacy single-core application software by transforming the code to eliminate dependencies between individual functions. A functional redesign and parallelization of compute intensive parts is ideally avoided and only attempted when resource constraints demand even more parallelism. The second activity is to implement the exposed parallelism using specific implementation patterns that guarantee time- and value-deterministic parallel execution on the targeted multi-core processor. We observed that most of these patterns had coordination regimes in common where state variables are updated and buffered at controlled instances in time, typically at the beginning or at the end of periodic tasks. These patterns were typically deeply project and supplier-specific and often lacked formal foundations. An automotive OEM has to consider that significant amounts of OEM-owned application software is integrated as part of a supplier-owned execution platform. It is mandatory for the OEM that changing a supplier does not result in having to redesign the application software due to supplier-specific implementation patterns. Therefore, when we found that many implementation patterns shared key concepts of the LET programming model [1] we set out to evaluate seriously whether (1) LET could be an efficient and effective basis for implementing parallel execution of periodic control functions and (2) whether such an implementation could be integrated with the runtime facilities of an important automotive software standard AUTOSAR [2]. If successful, the LET paradigm would also offer the added advantage to extend naturally to the coordination of time triggered event chains which are distributed across multiple interconnected ECUs. Predictable behavior of such distributed event chains is becoming more and more important with the growing sophistication of vehicle control functions.

The main contribution of this paper thus consists in describing first significant steps towards LET implementations for multi-core architectures using facilities of the automotive industry standard AUTOSAR. After describing a legacy powertrain control application, we sketch our approach to expose its parallelism. Then we outline the steps taken to implement the exposed parallelism using the Timing Definition Language (TDL) as an implementation of the LET programming model. We conclude with some preliminary results and a summary of next steps.

## II. Related Work

The quest for achieving predictable behavior in embedded systems is not new [3]. Embedded multi-core processors aggravate this problem to the point where rigorous restrictions at all levels of system design are required in order to have any hope for success [4]. The LET paradigm has been designed from the beginning to be a programming model that provides a basis for predictable behavior both at the design and at the implementation level [5]. The LET paradigm was and typically

still is met with healthy skepticism by the real-time and embedded community because of its underlying restrictions concerning general expressiveness and WCET estimate requirements and the hard to afford computational costs regarding buffer space and execution time in particular. But a recent application of the LET programming model to an industrial engine control software for a single-core execution platform has shown that these overhead costs can be controlled [6]. [6] also illustrates that a process for applying the LET paradigm exists by which legacy software can be transformed iteratively. At the same time, LET gains popularity again as a design principle to analyze and engineer real-time control systems [7]. Existing AUTOSAR timing services are still insufficient to support the LET model. Elements of LET were behind an early AUTOSAR concept proposal "Support for Predictable Software Execution" [8] but this concept eventually didn't receive enough support to be pursued further. If the experiment we describe in this paper is successful, we hope to revive this AUTOSAR community effort.

## III. Powertrain Control Case Study

The subject of our LET based parallelization effort is the central coordinator of an electric powertrain. Its core responsibility is command and control of a predictive operating strategy for the components of an electric powertrain: inverter, battery, auxiliaries, vehicle interface. A supplier is responsible for the ECU hardware and the AUTOSAR basic software (BSW) as well as for functions that directly interface with the ECU hardware. The supplier also provides an AUTOSAR compliant execution environment for integrating the application software (ASW). The ASW itself is developed by Mercedes-Benz and consists mainly of torque coordination, energy management, thermal management, auxiliary management, prediction and monitoring. The C functions that implement these features are called from a single fixed-period OS task. A reduced call rate is hard-coded in the OS task for each function whose period is a multiple of the base period. At the beginning of the task, task-external input from sensors and from the attached networks is processed, then the application functions are executed in a predefined static order, and finally output signals are processed and propagated either to their respective actuators or to the network. I/O itself is carefully coordinated either synchronously (polling) or asynchronously (interrupt based). This is a proven and pervasive software execution pattern for single-core embedded control units from which we would not want to deviate were it not for the fact that in the not too distant future single-core performance will no longer suffice for implementing advanced and innovative powertrain control features. We are thus faced with the typical exercise of parallelizing legacy software: partition the main task into independent subtasks which can be executed in parallel without affecting the overall functional correctness that is partly incorporated in the static execution order and the explicit or implicit dependencies between the individual functions.

## IV. Towards Parallelization

To achieve the most possible parallelization in our legcy software, we present a method which is mainly an application of [9]. Neither the functional architecture nor the software architecture of the legacy code was designed to support a distribution on cores of a multi-core processor. Nevertheless, since the code consists of a large number of 'elementary' software modules, or runnables, the dataflow architecture should be rich enough in structure to identify sufficient parallelism among the runnables to foster a possible distribution on cores. The dataflow architecture is represented as a graph, consisting of nodes identified as the runnables, and directed edges identified as global variables for data exchange among runnables. Assuming that runnables are mapped onto periodic tasks for implementation, forward dependencies represent data exchanges updated in the same cycle of the task, whereas backward dependencies represent variables updated during the next cycle of the task. The goal is to maximize inherent parallelism by identifying subsets ('clusters') of totally independent runnables. Manipulation of the graph are made possible, by allowing backward dependencies to be changed to forward dependencies while maintaining all forward dependencies. Concretely, the method consists of the following three steps (cf. Fig. 1):

1.) For the set of runnables mapped onto a periodic task, the corresponding dataflow graph with forward and backward dependencies has to be identified.

2.) In application of the procedure given in [9], with the additional requirement that there are no forward and no backward dependencies among runnables of a cluster, we start with the identification of 'starting nodes' of the data flow graph. A starting node is a node with no dependent predecessor nodes. After this first step, all edges from the runnables in this cluster are removed such that a set of new starting nodes emerge. This is continued until all nodes have been visited. From inspection of the first part of Fig. 1 we see that runnables R1, R2 and R7 satisfy our conditions and define therefore our first cluster. In the next step we remove all edges from runnables R1, R2 and R7, and find the next subset of starting nodes. These are runnables R3 and R9. Going to all nodes we eventually arrive at a clustering shown in the second part of Fig. 1. This procedure can of course be done manually as well as automatically.

3.) Since the runnables in each cluster are totally independent, the chronological sequence in which they are executed is arbitrary, and in particular they can be executed in parallel. These sets are then subject to distribution on cores of a multi-core processor, as shown in the third part of Fig. 1, where a distribution on two cores is shown. This distribution of the independent runnables should in further steps be subject to more considerations, e.g., measured executions times of the runnables on a certain processor, number of dependencies among different cluster and thus generated cross-core overhead, functional and non-functional requirements, and further system-level considerations.

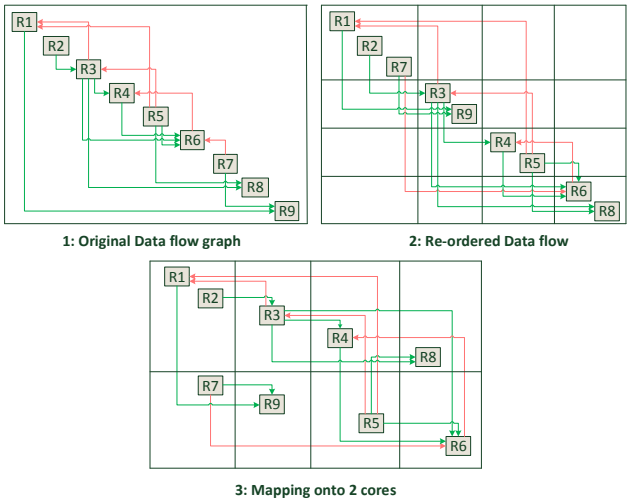


Figure 1. Parallelization with Data Flow Graph

## V. TDL for AUTOSAR Parallel Legacy Software

TDL allows specification and implementation of timing properties of real-time applications according to the Logical Execution Time (LET) paradigm. In the LET programming model, a fixed logical duration is associated to each execution of a computational unit, or task [1]. The inputs of the task's execution are those available at the LET start and the outputs of the task's execution are made available at the LET end. Thus, the LET model achieves a pre-specified, platform-independent observable temporal behavior of a set of software functions, leading to both time and value determinism [5].

The top-level unit in TDL is called a module, which contains declarations of sensors, actuators, tasks, and modes. Sensors and actuators model data sources and sinks, respectively; they are employed to communicate with the environment. A task declaration specifies an application function as well as corresponding input and output ports. A mode is a periodic sequence of activities: task LET instances, actuator updates, and mode switches. Mode activities are carried out by a runtime system (virtual machine) called E-machine. More details about TDL can be found in [10].

### A. TDL Modeling of Legacy Systems

TDL is accompanied by a commercial tool suite that can be integrated in top-down, model-based development processes, as well as in bottom-up, legacy-based development. The tool suite supports four development stages, which are described for the legacy case below.

1) *Programming:* The TDL program can be automatically generated from legacy information about the application functions associated with TDL tasks such as: connectivity, execution sequencing, periodicity, execution time (WCRT), and distribution on cores in a multi-core system, or on nodes in a distributed system. The resulting TDL program satisfies the legacy constraints (e.g., execution sequencing is preserved) and may include default LETs generated according to a user-specified policy (e.g., LET=WCRT). The user may then manually adjust LET values, e.g., by increasing some LETs in order to increase robustness against future additions of new functionality.

2) *Distribution and scheduling:* Mapping of TDL modules to computational nodes and cores is performed. Network communication schedules can also be generated in this step.

3) *Code generation:* The following runtime components are generated.

- The timing code, also called E-code, is compiled from the TDL program. The E-code is interpreted by the E-machine at runtime, triggering executions of LET start and LET end operations.

- Functions for implementing LET start and LET end operations for each TDL task, also called LET drivers. In order to achieve the LET data transfer semantics, certain legacy variables may need to be buffered. Input variables are buffered at the LET start of a task, while output variables are buffered throughout the execution and updated at the LET end. Optimization algorithms are put in place for minimizing the buffering.

- Setter/getter functions for the legacy variables that are subject to buffering requirements.

4) *Integration:* The integration of TDL components affects all levels of the legacy system.

- The E-machine employs some timing measurement capability of the hardware - a programmable timer, typically. The E-machine is executed within an interrupt service routine triggered by the timer.

- LET drivers are included in dedicated high-priority OS tasks, synchronized with the E-machine. Each LET driver task is also synchronized with OS tasks containing TDL-modeled legacy functions.

- The generated setter/getter functions are integrated at the application level.

### B. TDL Modeling of Parallelized Powertrain Control Software

Consider a legacy application with a parallelization structure as described in section IV. Assume that a cluster contains runnables with the same execution period (otherwise split the clusters accordingly). We propose the following TDL-based procedure for deploying the application on a platform with $n$ cores. First, assign to each cluster a number of $n$ TDL tasks (one per core) with the same LET, then group the runnables of the cluster into $n$ sequences and associate each sequence to a TDL task. If execution time information is available per runnable, then this grouping can be done such that the resultant LET value is minimized, which means core load is balanced within the LET and data propagation delay is minimized. In general, other (non-TDL) criteria may be considered in the assignment of runnables to cores - for example, one may wish to reduce inter-core communication by allocating runnables from different clusters with intensive data transfer to the same core.

The TDL program is generated such that the task's LETs are sequenced according to data dependencies between clusters. A TDL module contains then all the TDL tasks allocated

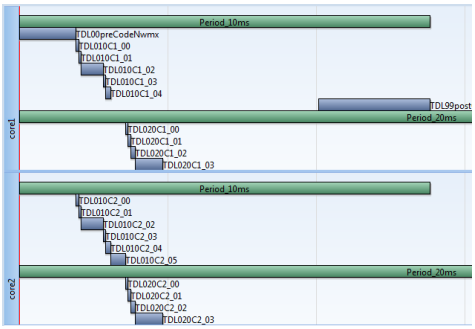


Figure 2. Visualization of a TDL program example

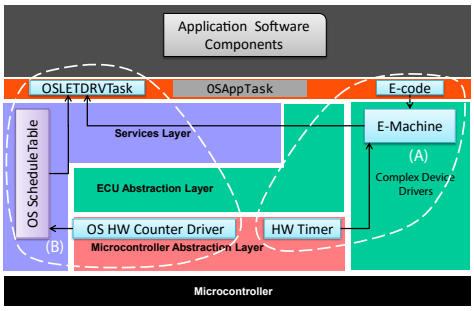


Figure 3. Integration of TDL in AUTOSAR with the two alternatives for the E-machine

to the same core. In the second stage, the module-to-core mapping is done by specifying the actual cores. In the third stage, a buffer analysis step determines a minimal number of legacy variables that must be buffered. In general, this is the case for communication variables between concurrent TDL tasks with overlapping LETs. Fig. 2 shows an extract from a TDL program, consisting of TDL tasks with periods 10ms and 20ms. The full program has 31 TDL tasks mapped to two cores, with $LET = 3 \cdot WCET$, plus rounding for alignment. The number of variables for inter-task communication exceeds 1500, of which only 7 need to be buffered.

In the fourth stage, integration of setter/getter functions is easily done by re-defining the implementations of ports and connections between AUTOSAR software components. LET drivers and E-code are included in the RTE. AUTOSAR OS events are employed for synchronization between the E-machine, the LET driver OS tasks, and the legacy OS tasks.

We are currently investigating two ways of integrating the E-machine into an AUTOSAR environment: (A) as a complex device driver (CDD) and (B) via OS schedule tables - see Fig. 3. The E-machine as CDD represents an additional basic software component that is able to deal with TDL programs of arbitrary complexity (in terms of modal structure and number of TDL tasks). The E-machine as OS schedule tables requires no additional AUTOSAR service or interface, but scalability is a concern that needs to be further investigated.

## VI. Conclusion and Future Work

This paper describes an approach for distributing single-core legacy software on a multi-core platform, where the application source code is kept unchanged by using runnables as the granularity for distribution. In order to ensure functional correctness, dataflow constraints across parallel execution threads are guaranteed by employing a multi-core implementation of the LET programming model. This is provided by the Timing Definition Language (TDL) and its tool suite. First analysis results indicate that the TDL overhead can be controlled to acceptable levels. Furthermore, we describe primary concepts for integrating the LET paradigm within AUTOSAR.

In the next steps, we will evaluate TDL runtime overhead and resource usage on a prototype system. We will investigate policies for choosing LET values, and for dealing with LET exceptions (violations of LET specifications). Further research will extend the approach to distributed functions, with components running on different nodes of a network (e.g., FlexRay or Ethernet).

Special attention will be dedicated to paving the way for LET standardization as part of AUTOSAR. We consider that there is enough room for different LET implementation techniques, in keeping with the AUTOSAR motto ”cooperate on standards, compete on implementation”. The main challenge is to gather convincing evidence that LET addresses a variety of use cases, involving different HW configurations (single- and multi-core, networked ECUs), software development processes (model-based design, including legacy software, simulation and testing, debugging), and industry players (OEMs, system suppliers, HW vendors).

## References

[1] T. Henzinger, B. Horowitz, and C. Kirsch, “Giotto: A time-triggered language for embedded programming,” *Proceedings of the IEEE*, vol. 91, pp. 84–99, January 2003.

[2] AUTOSAR, “AUTomotive Open System ARchitecture,” http://www.autosar.org/, accessed: 2016-01-11.

[3] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange *et al.*, “Building timing predictable embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4, p. 82, 2014.

[4] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, “Predictability considerations in the design of multi-core embedded systems,” *Proceedings of Embedded Real Time Software and Systems*, pp. 36–42, 2010.

[5] T. A. Henzinger, “Two challenges in embedded systems design: predictability and robustness,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 366, no. 1881, pp. 3727–3736, 2008.

[6] S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree, “Applying real-time programming to legacy embedded control software,” in *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 2015, pp. 1–8.

[7] D. Ziegenbein and A. Hamann, “Timing-aware control software design for automotive systems,” in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 56.

[8] C. Aussaguès, “Deterministic and dependable (also known as predictable and robust) embedded real-time systems… …with the OASIS and PharOS technology,” 2012, invited Talk at the 17th IEEE International Conference on Engineering of Complex Computer Systems.

[9] T. C. Hu, “Parallel sequencing and assembly line problems,” *Operations research*, vol. 9, no. 6, pp. 841–848, 1961.

[10] J. Templ, “Timing Definition Language (TDL) 1.5 specification,” University of Salzburg, Tech. Rep. T024, July 2009, http://www.softwareresearch.net.