Applying Real-time Programming to Legacy Embedded Control Software

Stefan Resmerita*, Andreas Naderlinger*, Manuel Huber†, Kenneth Butts‡ and Wolfgang Pree*
*University of Salzburg, Software Systems Center, 5020 Salzburg, Austria
Email: {stefan.resmerita, andreas.naderlinger, wolfgang.pree}@cs.uni-salzburg.at
†Fraunhofer Research Institute AISEC, 85748 Garching, Germany
Email: manuel.huber@aisec.fraunhofer.de
‡Toyota Technical Center, 1555 Woodridge, Ann Arbor, MI 48105, USA
Email: ken.butts@tema.toyota.com

***Abstract*—In the Logical Execution Time (LET) programming model, fixed execution times of software tasks are specified and a dedicated middleware is employed to ensure their realization, achieving increased system robustness and predictability. This paradigm has been proposed as a top-down development process, which is hardly applicable to a large body of legacy control software encountered in the embedded industry. Applying LET to legacy software entails challenges such as: satisfying legacy constraints, minimizing additional computational costs, maintaining control quality, and dealing with event-triggered computations. Such challenges are addressed here by a systematic approach, where program analysis and modification techniques are employed to introduce efficient buffering into the legacy system such that the given LET specifications are met. The approach has been implemented in a tool suite that performs fully automated transformation of the legacy software and may be carried out incrementally. This paper presents an application to large-scale automotive embedded software, as well as an evaluation of the achieved LET-based behavior for industrial engine control software.**

Keywords*—real-time; legacy software; logical execution time; middleware;

I. Introduction

Software applications for industrial embedded control systems have been traditionally developed with focus on achieving fast response times with minimal hardware resources. Thus, many legacy systems are subject to optimization for execution time and memory, which has been achieved by employing certain programming patterns, as well as by applying platform configuration and code compilation techniques.

Higher software complexity, together with faster and cheaper hardware has led to a shift in the importance of requirements towards other concerns such as robustness, predictability, safety and Quality-of-Service issues. In order to deal with such properties, execution time of software should be considered as a first-class citizen in the design process [1]. In this respect, we consider here the Logical Execution Time (LET) model [1], which is employed in several timing specification languages and tools such as: Giotto [1], xGiotto [2], the Hierarchical Timing Language (HTL) [3] and the Timing Definition Language (TDL) [4]. The LET paradigm assumes a software system consisting of a set of periodic and concurrent software functions having only well-defined data dependencies among them, and specifies for each function a fixed start time and a fixed end time for the execution of the function within its period, representing the logical execution time of the function. In fact, the LET semantics focuses on the time instants when data transfer between functions may take place, not on the actual physical execution of the software. LET-based approaches have a variety of advantages, in particular they provide timing predictability and increased robustness of the embedded software application [5].

In this paper we present an application of the LET programming model to industrial-level legacy automotive control software. The main technical challenge of such an endeavor is to reconcile the performance-oriented concerns of the legacy system with the non-functional requirements of the LET model. On the one hand, introducing LET behavior incurs additional computational costs and may lead to a decrease in control quality (mainly due to additional delays in data transfer among control functions). Thus, one has to minimize the required platform resources and the impact on the runtime performance of the system. On the other hand, the legacy software does not satisfy some assumptions made in the classical LET-based programming. In particular, many legacy applications use shared memory (global variables) for inter-task communication, while the LET model implicitly assumes function-scoped variables.

We outline a systematic approach where, given a logical timing specification for some of the time-triggered (periodic) control functions, a transformation is applied to the legacy software such that any execution of the transformed software satisfies the LET specifications. No change is made to the original mapping of functions to OS tasks or interrupt service routines. Special attention is devoted to reducing the computational costs of the transformed system. Our approach has been implemented in a tool set which performs the software modifications automatically. This has been applied to a large-scale legacy automotive control application with results that have exceeded expectations: low computational costs, no decrease in control quality, and observable increase in timing robustness.

This paper is structured as follows. Section II briefly describes the LET programming model, its flavor of our choice - the TDL component model, as well as the type of legacy software considered in this paper - automotive control software. Section III presents our approach for applying the

LET programming model to such legacy software. Section IV deals mainly with evaluation results for a complex industrial application - engine control software, which are preceded by a brief description of our tool-chain. Section V discusses related work and Section VI concludes the paper and outlines future work.

II. Background

The context of our work comprises real-time programming and legacy software, as described next.

A. *The Standard LET Programming Model*

In traditional real-time systems, the time instants when software tasks provide their outputs are usually influenced by platform-related factors such as scheduling, system load and memory caches. To avoid this source of unpredictability, the *Logical Execution Time* paradigm [1] abstracts from the physical execution of a task and associates a logical execution time interval to each periodic top-level software function, also called a LET task. A LET interval spans from a *release* time instant to a *termination* time instant within the task's period. At the release time, all the inputs to the task are read and transferred to the task's *input ports*, and the *task function* is marked as ready for execution. The actual execution of the function may be delayed and preempted as decided by the underlying scheduler. The function reads input data only from the input ports, whose values remain unchanged (until the next release time). However, the physical execution must end before the task's termination time, when all the function's outputs are made available to the environment (which may include other LET tasks) via the task's *output ports*. 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].

Fig. 2 shows the main constituents of the LET programming model in a typical top-down software development process. The LET specifications are given as a timing program written in a specialized language (e.g., Giotto, HTL, TDL), which is independent of the execution platform and the implementation of task functions. A time-safety check (including schedulability analysis) must be performed to verify that the timing program can indeed be achieved for a specific platform. If so, the timing program is compiled into a set of special LET-scheduling instructions called E-code, which is interpreted at runtime by a virtual machine called the E-machine. The release/terminate operations are implemented in dedicated functions called *release/terminate drivers*. The E-code specifies which release/terminate drivers must be executed at which points in time.

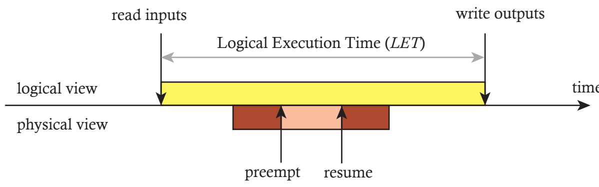

Fig. 1. The Logical Execution Time concept

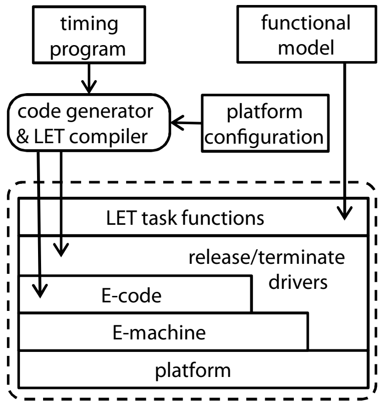

Fig. 2. Standard LET programming process

B. *The TDL Component Model for Real-time Systems*

We consider timing programs specified with the Timing Definition Language (TDL). TDL introduces a component model based on named containers called *modules*. A module encapsulates an automaton consisting of a set of modes, each representing a particular operational state of the software system. A mode defines a set of tasks with their LETs and interconnections, to be concurrently executed whenever the mode is active. A mode also specifies guarded transitions to other modes in the module. At runtime, all modules of an application run logically in parallel, each module having at most one active mode at a time. The time-safety check mentioned above ensures at compile time that adding a new module does not affect the overall system behavior. Additionally, TDL's *transparent distribution* property [6] ensures that the behavior of a distributed application is independent of the mapping of modules to computational nodes of a communication network. This is achieved by subsuming both computation and communication times under the LET. While this paper presents a single-node application, we take advantage of TDL's ability to deal with event-triggered functions and its flexibility with respect to specifying a task's LET within the task's period [4].

C. *Automotive Legacy Control Software*

Legacy software for engine and powertrain control contains both time-triggered and event-triggered functions, on top of a real-time operating system with fixed-priority preemptive scheduling, such as OSEK or AUTOSAR OS. The interrupt service routines and some event-triggered tasks have higher priorities than the time-triggered tasks. OS-level scheduling is complemented by application-level scheduling of periodic control functions, which are usually grouped into a small number of OS tasks with different priorities and activation periods. Thus, functions with the same priority are executed within the same OS task and the execution periods of these functions are multiples of the task's period. The functions communicate via shared memory. Event triggered tasks are mainly employed for state estimation. Calculation of control outputs is performed in the time-triggered part.

Illustrative example: We introduce here a running example, to be used throughout subsequent sections of the paper for illustration purposes. Consider a legacy system that includes five operating system tasks with task functions

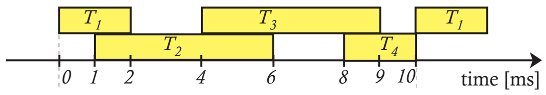

Fig. 3. Logical execution times for the running example

$f_i, i = 1, \ldots, 5$, as follows. Four of the tasks are periodic: f_1 and f_4 are high priority tasks with a period of $10ms$ each, while f_2 and f_3 have lower priority and a period of $20ms$ each. Task f_5 is event triggered, being activated by an external interrupt. Its priority is higher than that of f_4 but lower than that of f_1. The tasks communicate via two global variables a and b. Variable a is written by f_2 and f_5, and read by f_1, f_2, f_3, f_4, and f_5. Variable b is written by f_4 and read by f_3. The LET requirements that must be observed by the system are as follows. Tasks $T1$ and T_4 have a period of $10ms$ and a LET of $2ms$ starting at offsets 0 and $8ms$, respectively. Tasks $T2$ and $T3$ have a period of $20ms$ and a LET of $5ms$ starting at offsets $1ms$ and $4ms$, respectively. These specification are given as a TDL program, which is omitted here due to space limitations. The corresponding LET arrangement is shown in Fig. 3.

III. Applying LET Programming to Legacy Software

The LET programming model has been proposed as a top-down development process for control software. As this is inadequate for application to legacy code as described above, we present in this section our work for bridging the LET-legacy gap. We describe first the general approach, followed by details of individual steps and ending with tool support.

A LET timing program associates a LET task T to a legacy periodic control function f_T with period $\tau(T)$, and specifies a release time $\phi(T)$ (as an offset relative to the start of the period), and a logical execution time $LET(T)$ such that $\phi(T) + LET(T) \leq \tau(T)$. Furthermore, the input- and output ports of a LET task T correspond to the input- and output variables of f_T. A program variable v is regarded as input-and/or output for f_T, if v is read and/or written by f_T or any function that is reachable by f_T, respectively. We say that a function f *reads* a program variable v if v occurs on a right hand side or in a conditional expression in the code of f. Also, f *writes* v if v occurs on the left hand side of an assignment instruction in the code of f.

The objective of a LET-based transformation of a legacy application is to ensure that the behavior of the control functions subject to LET specifications satisfies the LET semantics. We extend here this semantics to the case where the same variable is both an output of a LET function and an output of an event-triggered function. In this case, we restrict the output updates as follows: the output value computed by the LET function is assigned to the output variable at the LET end only if the variable has not been updated with a more recent value by the event function. This ensures that the system uses the most recent value and avoids undesirable update effects, as further discussed in Section III-D.

Fig. 4 outlines the main stages of the transformation from the original software (top-left in the figure) and desired timing program, to the LET-based software (bottom part). The timing

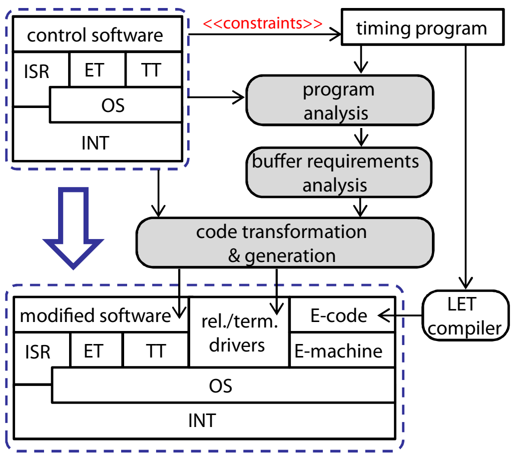

Fig. 4. Legacy LET transformation process

program is subject to constraints due to the requirements that the OS configuration must remain unchanged and modifications in application-level scheduling must be minimal. This is why, for example, the period of a LET task is inherited from the corresponding legacy periodic control function. The main stages are described below.

A. All the input and output variables of the LET functions are determined.
B. As implementing the task ports may require buffering, a buffer requirement analysis is performed in order to minimize the additional memory. This results in a set of new program variables to be added for buffering, called *add-on variables*.
C. A transformation of the legacy source code is done by instrumentation and/or changes of original lines of code such that, for every LET task T:
   - f_T starts execution no earlier than its release time in any execution period,
   - if an input port associated to an input variable p of f_T is buffered via an add-on variable p_T, then each read access to p is replaced by a read access to p_T in f_T and in all functions that may be called from f_T, and
   - if an output port for an output variable q of f_T is buffered via an add-on variable q_T, then each write access to q is replaced by a write access to q_T in f_T and in all functions that may be called from f_T.
D. A set of new functions representing the LET drivers is generated and included. Moreover, the E-code generated from the LET program and the generic E-machine are included with the transformed legacy software.

The above steps are described in more detail next, being also illustrated on the example introduced in the previous section.

A. Input/Output Variables

Function I/O variables are determined by using standard program analysis techniques, based on parsing the program's abstract syntax tree and control flow graph.

B. Buffer Analysis

The computational costs of the LET-based transformation are mainly determined by the number of add-on variables and by the code changes described above. While a realization that introduces a distinct add-on variable for each task port is indeed LET-compliant, the additional memory requirements and execution time (due to intensive buffering) may be significant for large legacy applications. However, not all ports need to be buffered; moreover, in some cases distinct port buffers may be implemented with the same add-on variable.

Our approach for reducing the number of add-on variables involves two stages:

1) Determine which task ports require buffers. This is done by a program analysis procedure which essentially checks the cases when using an unbuffered task port may lead to violation of LET semantics. For an input variable p of a LET function f_T, these are:
   a) p is written by a higher priority event-triggered function. In our running example (see Fig. 3), this is the case of variable a, which is an output of f_5 and an input to lower priority functions f_2, f_3, and f_4. This requires input buffers a^{in}_{T2}, a^{in}_{T3}, and a^{in}_{T4} to be used for a in the LET tasks T_2, T_3, and T_4, respectively.
   b) p is an output of another LET function f_U and a LET interval of T contains a termination time (LET end) of task U. For instance, input variable a of function f_3 is also an output variable of f_2. Thus, a buffer a^{in}_{T3} is required in $T3$ for a.
   c) p is an output of another LET function f_U with higher priority than f_T, the variable p is not buffered in task U, and a LET interval of U contains a termination time of task T. For example, the input variable b of function f_3 is an output variable of f_4 and it is not buffered in $T4$. Thus, a buffer b^{in}_{T3} is associated to b in $T3$.

   Let us see now what may happen when such a rule is ignored. Assume that we do not assign an input buffer to a in $T3$, so f_3 reads directly from a. Since a is an output of $T2$, a may be updated sometime in the interval $(4, 6)$ (e.g., if it is not buffered in $T2$), or at the latest in the terminate driver of $T2$, at time 6 (if it is buffered in $T2$). An execution of f_3 that reads from a anywhere between times 6 and 9 will violate the LET specification of $T3$ according to which the value of a used in the execution of f_3 must be the one at the LET start of $T3$, i.e., at time 4 in this instance.

   For an output variable q of f_T, the situations where not buffering q may lead to loss of LET compliance are as follows:
   a) q is read by an event-triggered function. For example, output variable a of f_2 is read by f_5, thus write accesses to a in f_2 must be done via a buffer a^{out}_{T2}.
   b) q is an input of another LET function f_U and a LET interval of T contains a release time (LET start) of task U. This is the case of variable a, which is output of f_2 and input of f_3, and there is a LET interval of $T2$ that contains a LET start of $T3$. This requires a buffer a^{out}_{T2} associated to output a in $T2$.
   c) q is an input of another LET function f_U with higher priority than f_T, the variable q is not buffered for task U, and a LET interval of U contains a release time of task T. For instance, variable a is read in f_1 (which has higher priority than f_2), it is not buffered in $T1$ and a LET interval of $T1$ contains a LET start of $T2$. Thus, also this rule requires the buffer a^{out}_{T2}.

   Note that the rules for the same direction (input or output) are independent. Thus, if an input test yields a buffer for input variable p in task T, the other input rules do not need to be checked for p and T. In our example, a^{in}_{T3} is required by two rules and a^{out}_{T2} is given by all three output rules. The buffering requirements for the example are: a^{in}_{T2}, a^{in}_{T3}, a^{in}_{T4}, b^{in}_{T3}, and a^{out}_{T2}.
2) Determine the add-on variables that are sufficient to implement the buffers. A straightforward way to allocate add-on variables is to use one variable per port buffer. However, the mapping from buffers to add-on variables also gives leeway for optimization in terms of memory requirement. Since a buffer is employed only for the duration of a LET interval, the same add-on variable may be used for different buffers (also associated to different variables) employed in non-overlapping LETs. However, reuse policies that minimize the add-on memory may require complex implementations, leading to the RAM gains being undone by the extra ROM and execution times of the implementation. As a trade-off, we have used a sharing policy that allocates a single add-on variable for all buffers of the same legacy variable in LET tasks with non-overlapping LET intervals. Furthermore, the same add-on variable is employed for the input and output buffers corresponding to the same legacy variable that is both an input and an output variable of a LET function.
   The add-on variables that are sufficient for implementing the buffering requirements for our example are: a_{T2_T4}, a_{T3}, and b_{T3}.

C. Code Transformation and Insertion

The purpose of legacy code modification is two-fold: replacing accesses to legacy variables with accesses to add-on variables in executions of LET functions whenever this is required, and inserting synchronization points between LET release drivers and the beginning of corresponding LET legacy functions.

As read and write accesses to legacy variables usually occur deep in the call hierarchy, in functions that are potentially invoked by multiple tasks or ISRs, a simple replacement of a legacy variable by a corresponding add-on variable is rarely applicable. A function f that accesses a variable v may be called from multiple LET functions, each of which may require an add-on variable in order to buffer v. If no add-on variable is required, then no change is done. The only case in which an occurrence of v in the code of f is replaced by an add-on variable v' is the one where v is read, f is never called from an event-triggered function, and v' is a common add-on variable for every LET task which calls f. In all other cases, the decision of which add-on variable to use instead of v (if any) is made at run-time. Thus, all read occurrences of v are replaced by calls to a reader auxiliary function (a getter function) and all write occurrences of v are replaced by calls to a writer auxiliary function (a setter). Each legacy variable with associated add-on variables has one getter and one setter deciding which add-on to use at each access, based on the current execution context (current LET task in execution). If the access happens outside the execution of a LET task, then the auxiliary functions select the actual legacy variable to be used.

Listing 1 Code changes for accesses to variable a

```
//before instrumentation
if (...) {
  a = local + 100;
} else if (a > 100) {...}
```

```
//after instrumentation
if (...) {
  LET_write_a(local + 100);
} else if (LET_read_a() > 100) {...}
```

Listing 2 Auxiliary access functions for variable a

```
int LET_read_a(void) {
  switch(current_LET_task) {
    case T2:
    case T4:
      return a_T2_T4;
    case T3:
      return a_T3;
    default:
      return a;
  }
}
```

```
void LET_write_a(int value) {
  switch(current_LET_task) {
    case T2:
      a_T2_T4 = value;
      a_T2_T4_dirty = 1;
      break;
    default:
      a = value;
      a_T2_T4_dirty = 0;
  }
}
```

To illustrate this with our running example, Listing 1 shows such transformations for part of a function that is called from both f_2 (time-triggered) and f_5 (event-triggered). The access functions for a are given in Listing 2 (as code in C).

As the results of the buffer analysis depend on LET overlapping of the tasks in the timing program, a modification of the overlapping (e.g. by adding new LET tasks or changing LET boundaries) is likely to change the existing buffer requirements. Such changes may be difficult to propagate in cases where legacy variables have been directly replaced by add-on variables. A fully incremental code transformation can be achieved if all accesses to the involved legacy variables are replaces by calls to auxiliary access functions.

For a LET function that is scheduled at the application level, synchronization with its release driver is necessary in order to ensure that the function starts execution no earlier than the LET start time in each period. In the case of OSEK, the synchronization is implemented with events: a $WaitEvent$ system call is inserted at the beginning of the LET function (before any initialization) and a corresponding $SendEvent$ is used in the release driver.

D. LET Drivers, E-code and E-machine

The release and terminate drivers are code-generated for every LET task. The release driver updates a task's input ports by transferring the value of each legacy input variable to the corresponding add-on variable (if this exists). Then it marks the corresponding legacy function as ready for execution. The termination driver updates a task's output ports by transferring the value of add-on variables (if they exist) to the corresponding legacy output variables. A LET task may have no assigned add-on variables (i.e., its input/output ports are implemented by the actual legacy variables), in which case no data transfer takes place in its drivers. The timing program is compiled into E-code and executed at run-time by the generic E-machine. This is implemented in a dedicated interrupt service routine triggered by a programmable timer interrupt.

Listing 3 Excerpt of LET drivers

```
void LET_release(int LET_task_id) {
  switch(LET_task_id) {
    case T2:
      a_T2_T4 = a;
      a_T2_T4_dirty = 0;
      break;
    case T3:
      a_T3 = a;
      b_T3 = b;
      break;
    case T4:
      a_T2_T4 = a;
      break;
  }
  mark_ready4_exec(LET_task_id);
}
```

```
void LET_terminate(int LET_task_id) {
  switch(LET_task_id) {
    case T2:
      if (a_T2_T4_dirty == 1) {
        a = a_T2_T4;
      }
      break;
  }
}
```

To deal with outputs of both LET and event-triggered functions, each add-on variable q_T corresponding to a LET task T and a legacy mixed output q has an associated "dirty flag", which is set whenever q_T is updated by the LET function f_T and is reset whenever q is updated by an event function f_E as well as in the release driver of T. Moreover, if q_T is shared with other LET tasks, then a distinct dirty flag is used for each task for which q is an output. In the termination driver of T, variable q is updated with the value of q_T only if the dirty flag is set. This prevents the situations where an update of q performed by f_E that takes place after the last update from f_T and before the LET end of T is overwritten in the termination driver of T (with the value from f_T).

Excerpts from release and terminate drivers for our running example are shown in Listing 3.

An execution example of the time-triggered tasks is depicted in Fig. 5, which also shows the access operations in the auxiliary functions and in the LET drivers. A situation where the event-triggered function f_5 preempts the execution of f_3 is shown in Fig. 6. Notice that the write to a in f_5 resets the dirty flag of a_{T2_T4}, thereby preventing the update of a in the termination driver of T_2. Thus, as opposed to the execution in Fig. 5, the value of a read by f_4 is not the one provided by f_2, but the one updated by f_5 (which is the most recent). This is similar to the behavior of the original legacy software.

IV. Tool Support and Evaluation

A. Tool Support

A tool suite has been developed in order to apply the transformation described above to complex industrial legacy software written in C with little or no manual intervention.

The program analysis was implemented in modules based on CIL [7]. This analysis yields the call graph and the list of global variables including all their read/write accesses. We have employed for the same purpose also the API of the Eclipse CDT tool to cross check the results. The algorithms for the buffer requirements analysis, and the mapping to add-on variables have been implemented in a dedicated Java-based tool. The same tool performs also the code generation of LET drivers and auxiliary access functions, as well as the modification and instrumentation of the legacy code.

B. Evaluation

The approach described in this paper has been applied to an industrial engine control software (ECS) application, consisting of hundreds of thousands of lines of C code. Thousands

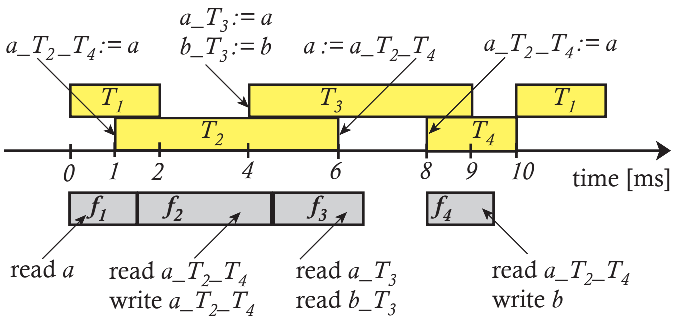

Fig. 5. Execution of time-triggered part: access and buffering operations

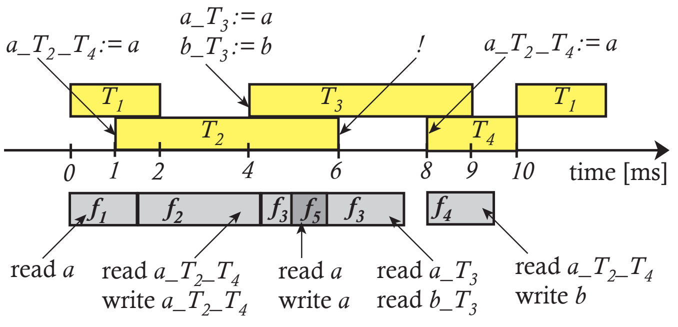

Fig. 6. Event-triggered execution with update of shared variable

of global variables were used in the ECS for communication between periodic control functions. The number of accesses to these variables in the entire ECS was over fifty thousand.

We focused on eight periodic control functions included in two OS tasks. Thus, the timing program referred to eight LET tasks, where the LET of a task was specified to be a little larger than the largest measured worst-case reaction time of the associated control function taken from multiple hardware-in-the-loop (HIL) simulations. This included the effects of event-triggered computations, most notably the crank angle computations, over the range of engine speeds.

The control software in these functions totaled more than 25% of the time-triggered software in the ECS. The number of basic type variables shared between the LET functions was about a thousand, and they occurred in about fifteen thousand lines of code in the whole ECS. Since the number of task ports was close to two thousand, the relatively straightforward approach to employ one add-on variable per task port was unacceptable. The first stage of buffer analysis (Section III-B) determined that buffering was needed only for about 30% of the ports. In the second stage, the number of add-on variables was further reduced by 15% (from the number of buffers), down to about 450. Thus, our approach achieved significant RAM memory savings in step (B) of the transformation, leading to an overall RAM increase of only 0.365%. Steps (C) and (D) led to an increase in the ROM memory of 1.3%. The additional execution times of the E-machine, LET drivers, and auxiliary access functions led to an increase in CPU utilization of 3.35% (absolute numbers are omitted due to intellectual property concerns). The execution time of our tool-chain on a desktop PC to analyze and transform the legacy software was in the range of minutes.

The behavior of the transformed system was evaluated relative to the original one by comparing state and output control signals obtained from HIL simulations, where ECS and LET-ECS were executed in turn on the same development ECU. This was run in closed loop with an engine model that was mostly simulated on a real-time computer except for some actuators which were physically present. A dedicated data acquisition system was employed to read (and record) program variables with a sampling period of 50 μs. In this section we use the term *signal* to refer to the sequence of values obtained by sampling one program variable during a HIL simulation.

Multiple same-input simulations led to slightly different signals for the same program variable; this was attributed mainly to the presence of physical parts in the control loop (which was a source of non-determinism). Thus, multiple signals were collected for the same software version, same variable and same driving scenario. Signals from ECS runs were compared with corresponding signals from LET-ECS runs using the mean relative error (MR) and root mean square error (RMS). Since each variable was updated periodically, the recorded variable values were downsampled with the variable's update period, and the errors were calculated between the downsampled signals. Moreover, jitter values were also compared.

Let S_O and S_L denote the sets of all signals obtained from the original ECS and from the LET-ECS, respectively. The signals $\overline{s_O}$ and $\overline{s_L}$ represent the means of the signals in S_O and S_L, respectively. For two signals $s = (s_1, s_2, \ldots, s_N)$ and $s' = (s'_1, s'_2, \ldots, s'_M)$ the mean relative error is $e_{MR}(s, s') = (1/n) \cdot \sum_{i=1}^{n} |(s_i - s'_i)/s_i|$ and the root mean square error is $e_{RMS}(s, s') = \sqrt{(1/n) \cdot \sum_{i=1}^{n} (s_i - s'_i)^2}$, where $n = min(N, M)$. We shall use $e(s, s')$ to refer to either error.

For a given program variable, the following error-based characteristics were used to compare the set of signals from simulations with the original ECS with the set of signals from the LET-ECS:

- Average: The error between the mean signals in the two sets, i.e. $e(\overline{s_O}, \overline{s_L})$.
- Worst-case: The largest error between any signal in S_O and any signal in S_L, i.e. $e^{OL}_{max} = max_{s \in S_O, s' \in S_L} e(s, s')$.
- Variation in the original set: The largest error between any two signals in S_O, i.e., $e^{O}_{max} = max_{s \in S_O, s' \in S_O} e(s, s')$.
- Variation in the LET-based set: The largest error in S_L, i.e., $e^{L}_{max} = max_{s \in S_L, s' \in S_L} e(s, s')$.

We say that a signal s fits into the set S_O if $e(s, \overline{s_O}) \leq max_{s' \in S_O} e(s', \overline{s_O}) + tol_k$, where tol_k is a tolerance depending on a small constant $k \in (0, 1)$. A signal from the LET-ECS that fits into the corresponding signal set from the original ECS is considered to be indistinguishable from the original signals.

Multiple simulations were conducted with both software versions, using driver inputs that took the engine through a prescribed speed profile. We include here the results for one state variable (engine speed) and one control output variable (throttle). The speed profile used in HIL simulations is shown in Fig. 7.

The error indicators for the two variables are given in Table I. The percentage of signals in S_L that fit into S_O is denoted by $r^{LO}_{fit}(k)$, for some tolerance tol_k, with $tol_k = k$ for e_{MR}

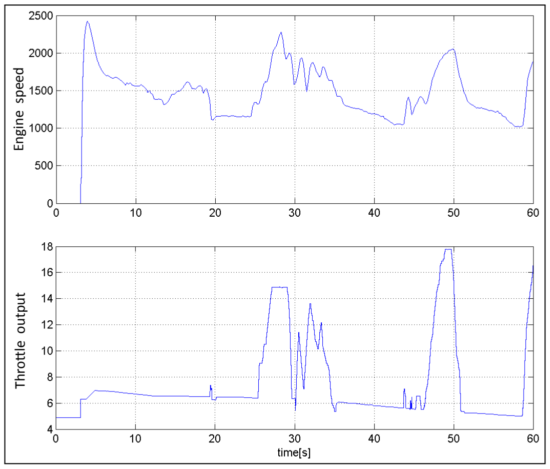

Fig. 7. Engine speed and throttle signals

TABLE I. VALUE COMPARISON

Measure	Mean relative		Root mean square	
	speed	throttle	speed	throttle
$e(\overline{s_O}, \overline{s_L})$	0.000077	0.003456	0.263555	0.037674
e^{OL}_{max}	0.001453	0.096657	5.104543	0.701864
e^{O}_{max}	0.001408	0.094881	4.995335	0.661189
e^{L}_{max}	0.001478	0.101022	5.113072	0.702298
$r^{LO}_{fit}(10^{-4})$	100%	72.22%	100%	72.22%
$r^{LO}_{fit}(10^{-3})$	100%	100%	100%	100%

and $tol_k = k \cdot mean(\overline{s_O})$ for e_{RMS}. The cardinalities of the signal sets are $|S_O| = 12$ and $|S_L| = 18$. Each signal has about $14,000$ samples.

From Table I one can see that the behavior of the two variables in the LET-ECS does not change from the original ECS: mean signals are extremely close, the largest error between any original signal and any LET-based signal is about the same as the largest error between any two original signals, and signal variation is also about the same. With a relative tolerance of 10^{-3}, every signal from the LET-based set is closer to the original mean signal than some original signal.

We compared the LET-ECS with the original ECS also from a timing viewpoint by looking at the cycle-to-cycle update jitter of state and output program variables. Consider a signal s representing variable v sampled over n periods; let t_i be the time when v is written during the i^{th} period relative to the beginning of that period. In other words, t_i is the time interval between the beginning of period i and the moment when v is written next. The jitter for s is calculated as the root mean square error between consecutive relative update times: $J_s = \sqrt{(1/n) \cdot \sum_{i=2}^{n} (t_i - t_{i-1})^2}$. We considered the mean and maximum jitter for each signals set. The results obtained for the above two variables are shown in Table II. One can see a significant jitter improvement in the LET-ECS.

The data for other state and output control variables shows the same pattern: the LET-ECS variables follow the same values as in the original ECS but with a better update jitter.

TABLE II. JITTER COMPARISON

Jitter measure	speed	throttle
$\overline{J^O} = (1/\|S_O\|) \cdot \sum_{s \in S_O} J_s$	1.08484E-04	1.09585E-04
$\overline{J^L} = (1/\|S_L\|) \cdot \sum_{s \in S_L} J_s$	2.73722E-05	4.17781E-05
$J^O_{max} = max_{s \in S_O} J_s$	1.80059E-04	1.58056E-04
$J^L_{max} = max_{s \in S_L} J_s$	3.11486E-05	8.20605E-05

This data shows that the LET-ECS works as intended and it is omitted here due to reasons of space. Most importantly, this behavior has been achieved with relatively small computational costs, demonstrating the applicability of our approach to complex industrial control software.

V. RELATED WORK

There are different dimensions of robustness in the context of embedded systems research, the most frequently encountered being fault tolerance. Robustness also includes the ability to sustain different kinds of variations in regular system conditions (e.g. WCETs, periods, system load or even hardware related properties) without severe consequences on the system behavior [8]. A general discussion on predictability, robustness and determinism in embedded systems can be found in [5].

Numerous papers propose mechanisms to achieve deterministic inter-task communication. For example, the works [9], [10], [11] are based on wait-free solutions and use buffering protocols with the common objective to minimize memory requirements. However, they follow a top-down approach where the concern is to preserve a certain behavior exhibited in the simulation while we impose new semantics to a legacy system. Moreover, they consider synchronous (reactive) semantics [12] while here we deal with LET-based behaviors.

LET was introduced in Giotto [1] and is the core programming model in several languages, such as TDL [4], HTL [3], xGiotto [2], and others. A detailed discussion on their differences can be found in [13].

AUTOSAR is now arguably the most prominent component oriented architecture in the automotive domain with the initial version of the standard being released about a decade ago. Only few papers address the growing calls for migration strategies of legacy code towards AUTOSAR, e.g. [14], [15]. AUTOSAR extensions for real-time requirements have been added only recently. A first concept of combining AUTOSAR with LET is presented in [16].

Reports of complex control software subject to LET specifications are scarce in the open literature. In [17], the autopilot software of an autonomous (unmanned) helicopter was re-engineered based on Giotto, with a LET timing program having three tasks with less than ten ports each. The software had no event-triggered functions. The only evaluation of the transformed system was in terms of additional execution time of the Giotto part. In [2] one can find an xGiotto-based implementation of a new fuel rate controller software with both event and time triggered tasks. In contrast to our setup, the software was developed top-down. No computational costs were given and the focus of the evaluation was on control performance of the new controller versus existing controllers (implementing different control laws). Here, the same controllers are used with and without LETs, enabling one to pinpoint the LET costs and benefits. In previous work [18], we applied the LET programming model to legacy control software under the restriction of making minimal changes to the legacy code, with the sole purpose of demonstrating the feasibility of achieving a LET-based behavior in such a complex setup. That approach made only code additions (instrumentation), and only in the top-level control functions. The downside was a high amount of extra memory (both RAM and ROM). In this work, we take a completely different approach, by making changes deep into the legacy source code, thus achieving the same LET-based behavior with acceptable computational costs. Moreover, all modifications are still confined to the application software: the ECU configuration, as well as the operating system remain unchanged.

VI. Conclusions and Future Work

The idea of introducing LET into legacy embedded systems is generally being met with skepticism by software and system engineers mainly due to the LET's perceived disadvantages: additional computational costs and degradation of control quality due to delays introduced in the time-triggered part between the moment when a control output is available and the moment when it is transmitted to the environment. This is compounded by a lack of methods and tools to enable quantitative evaluation of such costs against the LET-induced benefits.

This paper presents a systematic approach to LET-based software adaptation that makes it possible to perform such assessments with relatively little effort. The results undoubtedly depend on the legacy system as well as on the timing program that is desired. We describe in this paper a use case that is particularly challenging for LET: complex engine control software written in C, deeply embedded and highly optimized for performance, with application-level scheduling, shared memory communication, and high-priority event-triggered computations. The results confuted the customary pessimistic expectations: At a relatively small computational cost, and with no loss in control quality, the transformed software exhibited an observable increase in timing robustness, as shown by data obtained from hardware in the loop simulations.

Complex legacy software can be modified incrementally with our approach: one may start with a single LET task, test the transformed system, and then add more tasks if desired. From the viewpoint of our analysis, any function that is not called from a LET function is considered event-triggered.

The tool set is currently being extended to deal with pointer-based variable accesses. We plan to pursue further optimization of the buffer analysis in order to obtain buffering requirements that are necessary and sufficient. The tool set will be improved to provide the user with options for exploring trade-offs between RAM costs (number of add-on variables) and ROM plus runtime costs (incurred by code changes and instrumentation). As applications, we plan to extend the LET coverage of the ECS's periodic control functions. Other industrial applications are also foreseen.

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] A. Ghosal, J. Jurado, M. Sanvido, and J. Hedrick, "Implementation of AFR controller in an event-driven real-time language," in *American Control Conference, 2005. Proceedings of the 2005*, June 2005, pp. 4428–4433 vol. 7.

[3] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan, "A hierarchical coordination language for interacting real-time tasks," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*. New York, NY, USA: ACM, 2006, pp. 132–141.

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

[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] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," *SIGPLAN Not.*, vol. 40, no. 7, pp. 31–39, 2005.

[7] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 213–228.

[8] A. Hamann, R. Racu, and R. Ernst, "Methods for multi-dimensional robustness optimization in complex embedded systems," in *Proceedings of the 7th ACM &; IEEE International Conference on Embedded Software*, ser. EMSOFT '07. New York, NY, USA: ACM, 2007, pp. 104–113.

[9] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 21–33.

[10] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli, "An OSEK/VDX implementation of synchronous reactive semantics-preserving communication," in *OSPERT Workshop*, 2007.

[11] M. D. Natale and V. Pappalardo, "Buffer optimization in multitask implementations of simulink models," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 23:1–23:32, May 2008.

[12] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[13] A. Naderlinger, "Modeling of Real-time Software Systems based on Logical Execution Time," Ph.D. dissertation, University of Salzburg, 2009.

[14] D. Kum, G.-M. Park, S. Lee, and W. Jung, "AUTOSAR migration from existing automotive software," in *Control, Automation and Systems, 2008. ICCAS 2008. International Conference on*, Oct 2008, pp. 558–562.

[15] R. Anantharam and P. Kulkarni, "Methodology for migration of traditional application software to AUTOSAR architecture," ser. SAE Technical Paper, Detroit, 2014, SAE Technical Paper 2014-01-0191.

[16] M. S. Vladimir Belau, Hermann von Hasseln, "Coordinating AUTOSAR runnable entities using giotto - first concepts," 2012, poster presented at DEPCP 2012, March 16, Dresden, Germany.

[17] C. M. Kirsch, M. A. A. Sanvido, T. A. Henzinger, and W. Pree, "A giotto-based helicopter control system," in *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, 2002, pp. 46–60.

[18] S. Resmerita, K. Butts, P. Derler, A. Naderlinger, and W. Pree, "Migration of legacy software towards correct-by-construction timing behavior," in *Proceedings of the 16th Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems*, ser. FOCS'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 55–76.