Timing Analysis and Code Generation of Vehicle Control Software using Taxys

Stavros Tripakis [a,1] Sergio Yovine [a,2]

[a] *VERIMAG, Centre Equation, 2 Av. Vignate, 38610, Gières, France. Tel: +33 (0) 476 634843. Fax: +33 (0) 476 634850.*

Abstract

PATH's automated vehicle control application software is responsible for the longitudinal and lateral control of each vehicle in a *platoon* [5]. The software consists of a set of processes running concurrently on a PC, reading data from various sensors (e.g., radar, speedometer, accelerometer, magnetometer), writing to actuators (throttle, brake and steering), and using radio to communicate data to other vehicles. The processes exchange data with each other using a *publish/subscribe* scheme. In this paper, we describe the current software, and propose a model written in the synchronous language Esterel [1]. We use Taxys [2,7], a tool for timing analysis of Esterel based on the Kronos model-checker [3], and the Esterel compiler Saxo-RT [6], to verify that the application meets its deadlines. Timing analysis is done on-the-fly during the execution of the appropriately instrumented C code generated by the compiler. Instrumentation allows the verifier to observe the execution time of the application code. The C code generated by Saxo-RT, appropriately linked to the *publish/subscribe* library, can be run on the vehicles.

1 Introduction

PATH's Advanced Vehicle Control and Safety Systems (AVCSS) project involves the design and implementation of automated vehicle control applications on a variety of vehicles. In this document, we focus on the platoon application where the architecture is responsible for controlling a set of cars moving autonomously in a *platoon* formation (one car behind the other, with a small distance, e.g., 4-6 meters, between them), on the highway and at high speed (e.g., 65 miles/hour). The supporting highway infrastructure consists of a sequence of magnets placed on the center of a lane (typically 1.2 meters apart).

[1] Stavros.Tripakis@imag.fr

[2] Sergio.Yovine@imag.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

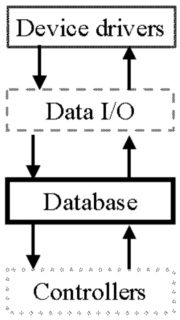

Fig. 1. Process types in the automated vehicle control software architecture.

The control functions can be divided into lateral and longitudinal control. The *lateral* control is responsible for keeping the car in the center of the lane, by reading magnet relative position information from the car's magnetometer and controlling the steering. The *longitudinal* control is responsible for maintaining a safe but short distance between the cars and for keeping the platoon stable. It does this by controlling braking and acceleration, using input information from the car's radar and other sensors, as well as information about the speed and acceleration of the car in front and the lead car of the platoon. This information is distributed among cars in the platoon using wireless communication. See [5] for more details.

First, we describe the software architecture of the above system, which consists of a set of processes running on the control computer (a PC) on each vehicle [4]. The processes include: *device drivers*, *controllers*, and *data I/O processes*. The device drivers interact directly with the hardware. The data I/O processes transform data from the device drivers into high-level C structures to be read by the controllers, and also transform high-level output data written by the controllers into low-level data for the device drivers. The controllers read high-level sensor data and compute high-level actuator data. The controllers interact with the data I/O processes via a *publish/subscribe* inter-process communication library. This is essentially a centralized database, providing to its clients (processes) the possibility to register/deregister, create/destroy variables, read/write variables, and ask to receive notifications when a variable is updated. Currently, the software is written in C and runs on the QNX real-time operating system. Figure 1 shows the interaction between the different types of processes and the database.

The second objective of the paper is to study the timing properties of the application software. In particular, we are interested in verifying whether the system meets its real-time requirements, in terms of *deadlines*. To do so, we first re-write the application in the synchronous language Esterel [1]. We then use Taxys [2,7], a tool for timing analysis of Esterel based on the Kronos model-checker [3], and the Esterel compiler Saxo-RT [6], to verify that the application meets its deadlines. The C code generated by

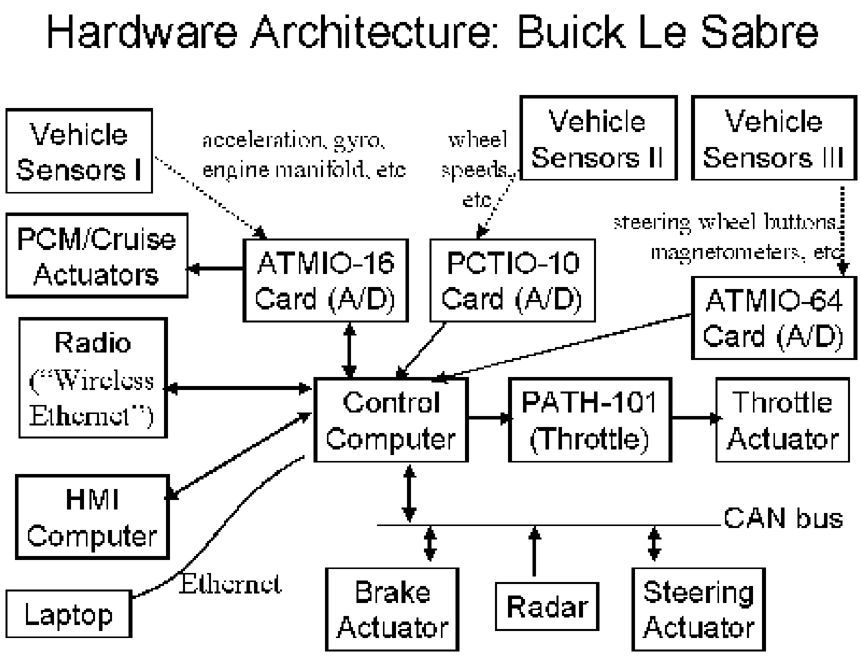

Fig. 2. Automated vehicle control: hardware architecture

SAXO-RT, appropriately linked to the *publish/subscribe* library, can be run on the vehicles.

2 Hardware architecture

The hardware architecture is shown in Figure 2. The boxes represent different pieces of hardware. The arrows represent connections of these pieces, and the direction of the arrows represents data flow: for example, the control computer takes input from the radar but not vice-versa.

The control computer is a 166 MHz Pentium PC. The "sensors" boxes I, II, III, are analog circuits taking inputs from accelerometer, magnetometers, and so on. The ATMIO-16, ATMIO-64 and PCTIO-10 cards are essentially digital/analog converter boards, equipped also with timers. PATH-101 is a card developed at PATH to control the throttle actuator. The other two actuators, brake and steering are connected to the control computer through a CAN bus, through which they receive control messages and send back status information. The radar (installed in the front of the vehicle) is also connected to the CAN bus and is used to report the distance of the subject vehicle from an object in front of it (typically vehicle-in-front). The laptop is used for initialization. The Human Machine Interface (HMI) computer provides status display to the passengers in the car.

3 The Publish/Subscribe library

The *Publish/Subscribe* library is used for communication between data I/O and control processes. It offers the service of a *centralized database* to a set of processes running on the same host.

The processes using the database are called *clients*. The database is a means for *asynchronous* inter-process communication, in the sense that a process producing data can write it to the database without worrying who the potential consumers might be, and at what pace they read the data. Consumers are also guaranteed to read the most recent value of data, which is of particular interest to control applications, where old data is often useless.

In addition to typical database operations the library also offers the possibility for clients to request to be notified whenever a variable is updated: these notifications are called *triggers* and can be seen as messages that are sent to a client process from the database.

In summary, the services offered by the publish/subscribe library are:

- Register/deregister with the database.
- Create/destroy a variable.
- Read/write a variable.
- Set/unset triggers for variables.

These requests are *atomic*, which means that the database will complete serving a request (receive the command, execute it, return the result) before it proceeds with the next request (that is, the database *serializes* the requests). Atomicity ensures in particular database *integrity*, for example, that the value read by a client is not modified during the reading process, and that a read always returns the most recent value of the variable in question.

4 Software architecture

Figure 3 shows the set of processes and their interactions. The device drivers are `pctio10` (PCTIO-10 card), `atmio16` (ATMIO-16 card), `atmioe` (ATMIO-64 card), `path101` (PATH-101 card), `cani` (CAN bus interface), and `radiodriver` (not shown in the figure).

The data I/O processes are the ones that deal with data acquisition, processing and output. They retrieve data from the device drivers, process it and store it in the database in a format that the control processes can use (i.e., C structures). They also retrieve from the database the control output produced by the control processes and write it to the device drivers. The communication between data I/O processes and the device drivers is *synchronous*. That is, the device driver blocks waiting for a read/write message from a data I/O process, receives such a message, process it by writing to the hardware, and replies back. One the other direction, some device drivers have associated interrupt handlers which get invoked whenever a hardware interrupt is raised

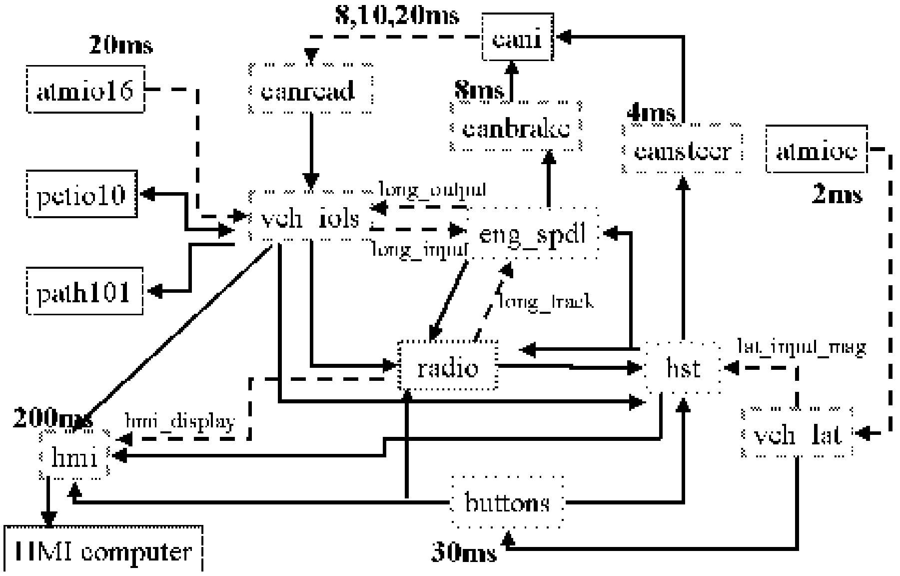

Fig. 3. Automated vehicle control software architecture.

by the device, and send an asynchronous (non-blocking) message to a data I/O process. The latter can then read data from the device. The data I/O processes are `veh_iols`, `canread`, `canbrake`, `cansteer`, `veh_lat`, `radio` and `hmi`.

The control processes are `eng_spdls` (longitudinal control) and `hst` (lateral control). The process `buttons` can also be seen as a control process, since it only interacts with the database. This process retrieves steering-wheel button activation data and current button status data from the database, computes new button status data and writes it back into the database.

Figure 3 also shows the variables exchanged by data I/O and control processes. These variables are actually created and stored in the database. Each arrow labeled with a variable means that the originator of the arrow updates the variable in the database, and the target of the arrow reads the variable from the database. Notice that there is a single producer for (process that updates) each variable. The exact information contained in the variables is not important for this document. For example, `long_radar` contains the range (in meters) to the nearest object in the front of the vehicle (presumably car in front), `long_brake` contains requested and achieved brake pressure, `long_input` contains acceleration (in meters/sec^2), engine speed (in rpm), and so on.

```
module veh_lat:
procedure
p_veh_lat()() ;
input atmioeI ;
output lat_input_mag ;
loop
  await atmioeI ;
  call p_veh_lat()() ;
  emit lat_input_mag ;
end loop
end module
```

```
module hst:
procedure p_hst()() ;
input lat_input_mag ;
loop
  await lat_input_mag ;
  call p_hst()() ;
end loop
end module
```

Fig. 4. Esterel implementation of `veh_lat` and `hst`.

5 Software implementation

All processes are implemented following the same pattern: an infinite loop which starts with a wait for a message; once the message is received, the process performs its function, and then goes back at the beginning of the loop. The source of the message can be either a timer or the database. Accordingly, we classify processes into *time-driven* (in fact, periodic) and *trigger-driven*. Time-driven processes wake up and perform their function periodically. In Figure 3, time-driven processes are labeled with a period in msec. Trigger-driven processes wait for triggers for one or more variables in the database. In Figure 3, each trigger-driven process has a dashed-arrow pointing to it, labeled with the name of the variable the process sets a trigger for.

We use Esterel [1] to prgram the application software. This language provides powerful constructs for management of parallelism and exceptions. It has rigorously defined semantics. Esterel programs run in a single thread on a single processor with a non-preemptive interrupt routine[3] and can refer to external data and routines written in C for complex (numerical) computations. Here, the application is decomposed into a control part, written in Esterel and a functional part written in C.

Figure 4 shows the Esterel implementation of the data I/O process `veh_lat` and the control process `hst`. `veh_lat` waits for a `atmioeI` signal corresponding to the periodic interrupt from the ATMIO-64 card (`atmioe`) every 2ms. Upon reception of this signal, `veh_lat` reads the ATMIO-64 device, and computes and updates the `lat_input_mag` variable in the database. This behavior is implemented by the C function `p_veh_lat()`. The update of `lat_input_mag` triggers a signal with the same name which unblocks `hst`. This process reads variables `lat_input_mag`, `lat_input_sensors` and `button_status`, and computes and updates variables `lat_output` and `marker_pos`, implemented by the C function `p_hst()`.

[3] The Esterel compiler eliminates parallelism by implementing a total order of execution consistent with the partial order defined by the Esterel program.

```
module veh_lat:                  module hst:
procedure p_veh_lat()() ;        procedure p_hst()() ;
input atmioeI ;                  input lat_input_mag ;
output lat_input_mag ;           loop
loop                               await lat_input_mag ;
  await atmioeI                    call p_hst()()
    %x:=age(atmioeI)% ;              %(325,345)%
  call p_veh_lat()()                 %deadline x≤2000% ;
    %(210,230)% ;                end loop
  emit lat_input_mag ;           end module
end loop
end module
```

Fig. 5. ESTEREL program augmented with timing information.

6 Timing information

In order to verify quantitative timing properties of the system, we need first to estimate the performance of the basic database primitives. Based on several experiments carried out, we found that:

- r - database read: 35μs.
- w - database write: 115μs.
- h - handle a hardware interrupt: 5μs.
- t - send a trigger event: 50μs.
- hw - hardware write: 50μs.
- hr - hardware read: 50μs.

Computations are esentially floating point operations, which are very fast. Therefore, computation time can be neglected [4].

Using the above numbers, we have, for instance, that process `veh_lat` takes approximately 220μs, corresponding to a hardware interrupt $h = 5\mu$s, a hardware read $hr = 50\mu$s, a database write $w = 115\mu$s, and a trigger $t = 50\mu$s.

Process `hst` takes approximately 335μs, corresponding to three database reads $3 \cdot r = 105\mu$s, and two database writes $2 \cdot w = 230\mu$s.

The timing constraint to be checked is that `hst` must finish before the next occurrence of the interrupt `atmioeI` from the ATMIO64 card. That is, the deadline for completion is 2000μs.

[4] In experiments we conducted, 20 million floating point operations took approximately 0.12 seconds on the 166 MHz Pentium machine. This averages to less than 1 microsecond for 1000 operations.

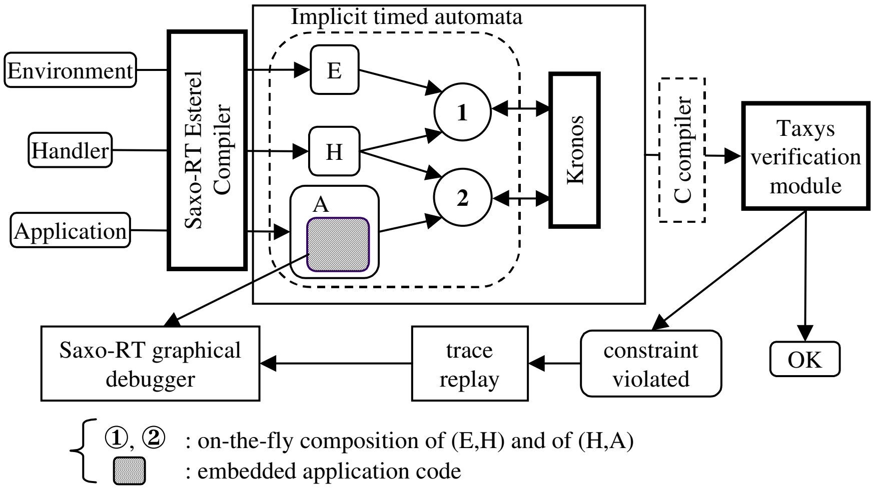

Fig. 6. Taxys.

7 Timing analysis

Timing analysis is carried out with the tool Taxys, jointly developed by VERIMAG and France Telecom R&D. Taxys is a tool for verifying quantitative real-time properties of Esterel programs. It is based on the Kronos model-checker, and the Esterel compiler Saxo-RT.

The Esterel code is augmented with the timing information. Figure 5 shows the corresponding code for the processes `veh_lat` and `hst`. Execution times are attached to the C functions. Intervals are used instead of exact values to account for the uncertainty in the timing information. When process `veh_lat` is waken up by the event `atmioeI`, the *clock* x is set to the time elapsed since the current instance of the event `atmioeI` has been emitted. The value of x is checked to be smaller than or equal to 2000μs, the period of the ATMIO64-card interrupts.

Periodic events are modeled as the environment of the application software. Figure 7 shows the Esterel module corresponding to the periodic emision of `atmioeI` by `atmioe` every 2000μs. The signal `TIME` and the procedure `time()` are provided by Taxys to model the passage of real time. To interface the program with its environment, we use an external event-handler which keeps track of the history of the environment and triggers computation steps of the application.

The architecture of the tool is depicted in Figure 6. The input files, namely the application and the environment, both written in Esterel with pragmas, and the specification of the event-handler, are compiled with the Esterel compiler Saxo-RT [6], which generates C code appropriately instrumented for performing timing analysis. The files are linked together with the Kro-

```
module atmioe:
output atmioeI ;
input TIME ;
loop
   await TIME ;
   call time()() %(2000,2000)% ;
   emit atmioeI ;
end loop
end module
```

Fig. 7. Periodic behavior of `atmioe`.

nos library to generate the Taxys executable verification module. Taxys performs symbolic forward reachability analysis *by running the instrumented code* and checks whether the specified deadlines are met. Notice that the timing verification is done on-the-fly during run-time. For this application, Taxys explored the entire reachable set (2022 symbolic states) in a few seconds and verified that all deadlines were indeed satisfied.

8 Code generation

The Esterel compiler Saxo-RT takes as input a set of Esterel modules like the ones shown in Figure 4 and generates C code that implements the Esterel semantics. External C functions such as `p_veh_lat()` and `p_hst()` are provided by the user. Assuming that the timing information used for the analysis above is correct, the resulting program is guaranteed to meet all the deadlines.

Correct timing information means that the execution times of external C functions on the target execution platform are withing the intervals specified above, e.g., `p_veh_lat()` takes between 210 and 230μsec to execute. Also, the environment behavior has to be as specified in the analysis, e.g., the signal `atmioeI` must be received every 2ms, as shown in Figure 7.

We can therefore replace the C code currently running on the vehicles by the C code generated by Saxo-RT, by doing the following. First, by providing the external C functions that implement the behavior of each module (reading from and writing to the database, and performing computations). Second, by linking the signals emitted by the environment (e.g., `atmioeI`) to the event handler associated with the Esterel program (the event handler is provided by the Saxo-RT suite). At the end, we have automatically generated and verified code.

9 Conclusion

We have presented the software architecture of a real automated vehicle control application, developed at PATH. We have proposed an implementation of the control software using the synchronous language Esterel. We have used the tool Taxys to verify that the C code generated by the Saxo-RT compiler would satisfy its timing requirements, provided the assumptions on the execution times of the platform-dependent C functions and the behavior of the environment (e.g., periods of signals) hold. Timing analysis is done on-the-fly during the execution of the appropriately instrumented C code generated by the compiler [7]. Instrumentation allows the verifier to observe the execution time of the application code. As a matter of fact, the C code generated by Saxo-RT, appropriately linked to the *publish/subscribe* library, can be used to run on the vehicles.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of computer programming*, 19(2):87–152, 1992.

[2] V. Bertin, M. Poize, J. Pulou, and J. Sifakis. Towards validated real-time software. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pp 157-164. Stockholm, Sweden, 19-21 June, 2000.

[3] C. Daws, A. Olivero, S. Tripakis, and S.Yovine. The tool Kronos. In Hybrid Systems III, Verification and Control, Lecture Notes in Computer Science 1066, Springer-Verlag, 1996.

[4] S. Tripakis. Software Architecture of PATH's Automated Vehicle Control. Technical report UCB/ERL M01/6, 2001.

[5] P. Varaiya. Smart Cars on Smart Roads: Problems of Control. *IEEE Transactions on Automatic Control*, 38(2):195-207, February 1993.

[6] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *Proc. CASES'2000*. San Jose, CA, November 17-19, 2000.

[7] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS: a tool for the development and verification of real-time embedded systems. In *Proc. CAV'01*, Paris, France, July 2001.