ReTiS: a Real-Time Simulation Tool for the Analysis of Distributed Real-Time Applications

Mikaël Briday, Jean-Luc Béchennec and Yvon Trinquet
IRCCyN, UMR CNRS 6597,
1, rue de la Noë - BP92101
44321 Nantes Cedex 3 - France

Abstract

This paper presents a simulation tool for real-time applications analysis. Simulation concerns the operational architecture and takes into account the executable code of real-time tasks and the execution support model: set of processors, network and basic software (real-time executive and communication system). We show in this paper a mechanism allowing to track data transfers and its interest for the analysis of the temporal behavior of real-time programs in a distributed system.

1 Introduction

The context of the work presented in this paper is that of real-time embedded systems. These systems, used for process control have to react on stimulus emitted by the process, and reactions have to respect the timing constraints of the process. During the design of a real-time system one has to cope with various architectures. First of all the “Software Architecture” describes the implementation of application functions and the “Hardware Architecture” represents all the hardware resources (processors and networks). But we need also two other architectures: 1) the “Technical Architecture” which is the association of Hardware Architecture and basic software (sometimes called Execution Support Architecture) and 2) the “Operational Architecture” which can be seen as the result of the mapping of the Software Architecture on the Execution Support Architecture. Software architecture is made up of a set of cooperative and concurrent activities (Event Triggered execution model). Execution Support Architecture consists, in the most general case, of a set of processors, a network and the basic software (real-time executive and communication system on every node). The work presented hereafter is at the Operational Architecture level.

Application timing constraints can be of various kinds: absolute or relative constraints, throughput constraints on inputs and outputs. They can concern simple treatments (i.e. maximum alarm response time) or a complete chain of treatments (end to end timing constraints). The missions of a real-time system can be critical. So if timing constraints are not met that can possibly have serious consequences on financial or human aspects. As a result of which the development of such systems requires the use of specific methods and techniques, particularly to prove the respect of timing constraints.

Verification of a real-time application is a huge and difficult problem. It must be led throughout the development cycle on functional and extra-functional aspects (temporal aspects, safety, ...). The Operational Architecture is needed for the verification of timing constraints because software and hardware aspects have to be taken into account. As long as the complexity of the Operational Architecture model - which depends on the abstraction level - remains low, formal techniques can be used to verify functional and sometimes temporal aspects [2], [4]. But as soon as one wants to take into account in a more detailed way the execution support, formal techniques are not appropriate anymore. Nevertheless, with semi formal techniques one can conduct a validation taking into account the temporal behavior of the real-time executive and the communication system [5]. Lastly, when the executable code is available one can carry out a validation using a very accurate model of the Execution Support Architecture. It is a step before the final test on the real target. All these various validation levels have different objectives. They are not conflicting: they are complementary.

Our contribution is related to the last phase of the development process, before the final test on target. It consists of the analysis of the execution of the code (application code and basic software) on a very detailed model of the Hardware Architecture. The used technique is that of simulation because the other analysis techniques would not be suitable. Our objectives are the temporal behavior analysis of the executable code on specific scenarios to verify timing constraints and to get time characteristics of the software in a multi-task, distributed environment while taking into account the operating system. Unlike classical debugging and analysis tool, neither instrumented code nor modified operating system is required, resulting in a
no time altered software. Interests of simulation techniques [3], [10], [1] at this step of the development - with regard to the final test on the target - are numerous:

- one does not need to have the target;
- one has no limitations of the final target: executable code in read-only memory, difficult control of task execution, difficult control of the environment behavior;
- analysis is not intrusive: no disturbance is brought;
- one can replay - indefinitely and without difficulties - delicate situations in order to study some problems on timing behavior;
- it is quite easy to set up observers for verifying that timing constraints are met, as well as to set up various time measurements.

The paper presents some results on the development of ReTIS (Real-Time Simulator), a simulator tool for Operational architecture analysis. It is organized as follows. First of all we present the modeling of an Hardware Architecture based on the Infineon C167 processor and its integrated CAN controller. This choice justifies itself by our applications in the field of automotive embedded real-time systems. Then we present a mechanism allowing to track the evolutions of a memory location. This mechanism is generic. It does not depend on the processor we use and it is the common denominator of all the checks which one can then realize on a real-time application: track of the values of an application variable, track of the task activations, track of the use of the task stacks etc. all of them being used for the verification of properties (timing or other types as buffer size).

2 Modelisation

2.1 Working environment

The modelisation and simulation platform environment is built on SystemC [11]. SystemC has been launched in september 1999 and is backed by a large community of over 45 companies from semiconductors, systems, IP, embedded software and EDA industries. SystemC is an Open Source object-oriented framework written with C++ that enables from Register Transfer Level (RTL) to system level design and simulation. A set of cooperating classes and templates allow to design boxes, ports and signals to link the boxes. Basic features of SystemC can be compared to those found in Hardware Description Language (HDL) like VHDL or Verilog but since version 2, SystemC offers high level features (channels, interfaces and events) which allow a higher level of design: the Transaction Level Modeling (TLM) [9]. SystemC has better performances than standard HDL too and so choosing SystemC as a working environment has many advantages:

- the models could be easily connected to models provided by hardware vendors.
- any C or C++ library can be included in a SystemC model.
- the availability of the source code allows to customize the simulation engine when needed.

2.2 Modelisation of the C167 core

The main core of the CPU consists of a 4-stages instruction pipeline, a 16-bit arithmetic and logic unit (ALU) and dedicated Specific Function Registers (SFR). Additional hardware is provided for a separate multiply and divide unit, a bit-mask generator and a barrel shifter. In the real-time context, it is necessary to model the CPU to reproduce not only the same behavior, but also the timing characteristics.

Two simulation approaches are attractive: The Instruction Set Simulator (ISS) only takes care of the behavior of the instruction, independently of the time needed to execute the instruction whereas a cycle accurate simulator takes into account timing of the real system (it models pipeline and jump cache for instance). Both simulation schemes are interesting. On one hand, a cycle accurate simulator is slow but essential for real-time systems simulation and on the other hand, ISS has better performances. To take advantage of both schemes, a modular simulator where the ISS is separated from the timing simulator has been designed. The ISS can run alone or connected to the timing simulator. The mode can be switched dynamically during the simulation (see [7]) giving us a very efficient way to quickly execute a large amount of code and then focus on a critical code execution window.

2.3 Simulating the CAN controller

The CAN protocol, created by Bosch in 1986 is widely used in automotive industry. It is based on the producer-consumer model. It’s a low-cost asynchronous multi-master bus with no shared memory, no traffic controller and no global clock. A node can send a message as soon as the bus is idle. A frame is composed of a message identifier that defines data (vehicle speed in km/h, motor rotation in rpm...) and up to 8 bytes of data. There is an enhanced error control for critical applications.

A message that is sent by a node is received by all the other ones (in case of no error), bit after bit. Then each node can apply a filter (on identifier) to select which message it is concern with, store the message and inform CPU that a new data is available.

As there is no global clock, each node resynchronizes itself on dominant (logical 0) to recessive (logical 1) edges. In case of monotonic sequence, the insertion of an opposite bit is necessary to resynchronize clocks (bit stuffing). As the number of bits added is data dependent, it is necessary to produce the whole message to know its exact length. It is not necessary to model the simulator at a
more accurate level (each bit is divided in time quantum), as electrical aspects are not considered.

The goal of the CAN simulator is to model a CAN components such as Philips SJA1000, Intel i527 or Infineon C167CR embedded controller. Each CAN component can be splitted in 2 parts, a specific part that manages message filtering, message storage and CPU interface, and a generic part that implements the protocol part (see figure 1).

2.3.1 Simulation of the CAN specific component using SystemC Transaction Level Modeling

The CAN generic component is a SystemC module that takes place between the CAN specific and the CAN bus components. This separation between CAN specific and CAN generic does not exists in the real component, the interaction is then realised with a standard C++ API that offers higher level functions than direct bit access such as sendMessage() or nodeError()

Since the goal is to model real-time behavior, the functional details can be changed to simplify the modeling. So a virtual arbiter which is implemented in the bus has been added. The SystemC Transaction Level Modeling new capability is used for this model. With the Transaction modeling, the communication between components are described as function calls (see figure 1) instead of logical signals as in a VHDL model.

Each node has only one bidirectional port. It is used to send orders to the bus (Request, nodeError, ...) and getting back information from the bus (receivedMessage, error, ...).

Each node that wants to send a message send a request to the bus. If at least 2 nodes request a bus transmission at the same time (i.e. in an interval less than a bus cycle), the bus arbiter selects the most important message (using arbitration field). It then detects when the other nodes lose the arbitration (they have not the highest priority) and send a corresponding message. At the end of the message, the bus inform each node that a message is present (receivedMessage).

Figure 1. Model of the CAN bus using SystemC Transaction Level Modeling. The CAN node can transmit directly a CAN message object to the node and not a series of bits.

Figure 2. model of the whole architecture

An error in a node or in the bus can easily be simulated with this model. Furthermore, all data traffic is centralised on the bus at message level (instead of bit level in a VHDL model). This model is also useful to attach other information into the CAN message (see section 3.2).

2.4 Simulating the whole architecture

In order to communicate with other devices such as CAN controllers, the processor model is wrapped into a SystemC module. The module has an input clock that is connected to a C++ method that simulate one cycle. Additional hardware can still be connected through external devices. Memory access are not modelised at the RTL (through pin WR, RD, CS, ...) because the CAN controller is on-chip. Anyway, the transactions are kept cycle accurate. The whole architecture consists of n C167 processors communicating through the CAN network (figure 2). Only one clock is used for all processors as the accuracy of the model is set on the CPU clock cycle.

3 Memory tracking

We call the technique presented hereafter memory location tracking. While low-level simulation is temporally accurate (when the hardware models are tuned enough), its main pitfall is the difficulty to extract high level informations from the data that was collected during the simulation. Memory1 tracking is based on a low-level mechanism (see 3.1) but it provides high level informations about how and when data are moved across the simulated system.

The first step is to define a variable that will be tracked during the simulation. Then, the simulation is started, and the simulator constructs a graph where each node is

1The term memory must be understood in the most general case, it can represent the main memory as well as general purpose registers or special registers.
Figure 3. Data Flow generated by the example in section 3. The temporary variable \( t \) is inserted by the compiler. In that graph, arrows represent a dated movement of data.

a memory address, and each edge represents a data movement (dependency relation between two variables). At the end of the simulation, the generated graph (data flow) contains useful information for debugging and analysis purpose:

- it is important, for example, to know that an information is available, but not used during a period because of a processor overload, which led to miss a deadline,

- one also has the information that the time constraint is not respected because another variable that brings a part of the result is not available on time,

- this technique can also be used with read-only memory. It records each program instruction access and one can efficiently detect how many accesses to a specific piece of code are performed, with time stamps.

Let's consider a simple example of algorithm (in C language\(^2\)):

```c
int a = 0, b = 0;
for(int i = 1; i < 3; i++)
{
    a = a + i;
    b = b + a * i;
}
```

\(^2\)The analysis performed by the simulator will be done on an assembly basis, as it simulates directly the final executable code, but the C language is more easily understood.

This simple example uses 3 variables (\( a, b \) and \( i \)) which are largely interdependent: \( a \) depends on \( i \) and that \( b \) depends on \( a \) and \( i \). The dependency graph (data flow) of that program can be drawn (figure 3).

In order to build a graph like this one, each read and write access to memory have to be detected and stored; then a link between the read access and its corresponding write access must be done. This part is the more complex one and can be done by an instruction (a mov instruction for instance), or peripheral components such as DMA (Direct Memory Access) or network components.

### 3.1 Detecting, storing and linking memory accesses

Each memory access (read, write or both) can be associated with an action (a piece of code that is triggered when the access is performed). It is very useful for interfacing the CPU with other components and for debugging purpose. To track a variable, a SpyAction that have in charge the tracking of all memory accesses at a particular address is used (see figure 4).

A SpyAction can also be associated to more than one address if the variable is more than 1 byte long. The information that have to be stored for each access is:

- the date of access. It is stored in number of cycles since the beginning of simulation,

- the type of access (READ or WRITE),

- the value of the variable,

- the length. It is possible to access only the first byte of a 2 bytes variable for instance,

- a mnemonic that gives information about the kind of access (this could be the mnemonic of an instruction such as MOV R1, R2 or an operation from a component such as CAN Msg 1, byte 3,

- One or more links to the corresponding access(es). A read access has a reference to its corresponding

![Figure 4. each memory address that have to be tracked is associated with a SpyAction. The SpyAction stores all memory accesses to that particular address in a list.](image)
3.2 Linking across the CAN network

When a node (the processor and the attached CAN controller) sends a message to one or more nodes, a vector that contains all the references to the Spy_Action associated to the memory locations of the send buffer is tied to the CAN frame (see figure 5). Each node that receives the CAN frame (after a possible filtering phase) gets this vector and establishes the links. A data read from the emitter can be linked up to \( n - 1 \) written data, with \( n \) being the number of CAN nodes), in the case of no filtering.

3.3 Linking across the instructions

As an instruction can perform more than one memory access, the main problem is to update our model by adding the semantic of the instruction; it is necessary to declare which accesses are logically dependent. Consider for instance the instruction "MOV [R1+], R2" that moves the data in register 2 indirectly to the address contained in register R1. It then increments R1. There are 2 read accesses (R1 and R2) and 2 write accesses (R1 and the address pointed by R1). It is obvious that the logical path of information is from R2 to the address pointed by R1 and R1 to R1. This must be taken into account in order to determine the actual dependencies.

4 Results

The variable tracking technique is used - obviously - with more complex applications than the simple example presented in section 3. The experimentations presented hereafter are based on a 3 nodes CAN network, each node running an OSEK/VDX operating system. The maximum of nodes that can be simulated in parallel is just limited by the amount of memory available. A network with 2048 nodes have been successfully experimented.

A graphical user interface has been designed using the Cocoa/GNUStep framework which is part of ReTIS. This interface can connect a remote (using a TCP/IP socket) or local simulator. The graphical interface is needed to display tracking graph in a more efficient way than a text-based approach.

4.1 The OSEK/VDX Operating System

OSEK/VDX OS is a multi-tasking preemptive kernel [8] which is used for embedded systems in the automotive industry and support time critical applications. Different conformance classes have been designed in order to support a wide range of hardware resources, with a high degree of modularity and ability for flexible configurations. Two types of tasks are supported by the kernel:

- **basic tasks**: They don’t have a waiting state and must terminate.

- **extended tasks**: they are allowed to enter a waiting state.

The task priority is statically defined. The resource management is done using the Priority Ceiling Protocol (PCP, [6]). For more informations about OSEK/VDX, one can refer to [8].

4.2 The benchmark application

The benchmark is based on a 3 nodes CAN network. Each node is an Infineon C167 with an embedded CAN controller, that runs OSEK-VDX (see figure 6). The application is different for each node:
Figure 7. Getting time interval a variable is waiting in the ring buffer. The variable is stored in the ring buffer (variable name is `fifo`) during 4892 µs (between date 5498756 and 647714). A “S” looking arrow between 2 variables (memory locations) corresponds to a data transfer. For instance, `Task1/producer1Value` is read at date 549335 and written in the variable `fifo` at date 549876.

Figure 6. Schema of the benchmark application.

**node 0**: The test program is a 3 basic tasks example based on a producer/consumer model. The system is configured in full-preemptive scheduling mode. Both writers and the consumer are periodic tasks. The writers’ priorities are 5 and 6 and the consumer priority is 7 (0 is the lowest priority). The period of the consumer’s task is the half of the period of the producing tasks in order to prevent an overflow of the buffer. The critical resource is a ring buffer with a priority set to 8 (rule of PCP protocol). As soon as a data is read by the consumer, it is sent to node 2 through the CAN network.

**node 1**: This program wait for a data from the previous node and computes a result. The data is retrieved using a polling scheme on the CAN component. The availability of the result directly depends on the reception of data from the node 0.

**node 2**: This program has the same function as the one in node 1, except that the data is retrieved using an interruption scheme on the CAN network.

Using this configuration, one can exhibit the variable tracking possibilities: getting very precise timing information about the use of a variable in the complete framework, including the multi-task operating system and the network. With the programs in nodes 1 and 2 having the same function, but implemented in different ways, quantified timing responses can be obtained.

### 4.2.1 Getting time information on a variable path

Before the simulation, tracked variables have to be defined, using literal or symbolic application name to reference the address(es). In this example, a variable that is produced in node 0 is tracked. This variable is transferred to the consumer through the ring buffer and then transferred to the node 2 through the CAN network.

During the simulation, all the memory locations that will be affected by this variable will be recorded. The simulation is started for 2 millions of cycles (100 ms with a 20 MHz C167), corresponding to a computation time of 3.8 seconds on a 2.8 GHz Pentium IV processor (for the whole architecture).
In order to limit the number of variable shown, only addresses that have an interest (no register and temporary variables) can be drawn in the ReTiS tool. As automatically and user created tracks are identified (section 3.1), one can select the data that will be shown. In this case the variable of the producer/consumer task of node 0, CAN registers and results variables of the node 1 and 2 are displayed (see figure 7).

In the graph, each line is associated with a pair (node number, address) and the time is in abscissa. When a data is read from an address and written to another one, an arrow is drawn, showing the data transfer. The date of access can be displayed (in cycles since the beginning of simulation). The date takes place in the lower part of the line for read accesses and in the higher one for write accesses. If different variable track are displayed, one can emphasize only one path. A zoom is also available to get time information: the time scale can vary from some cycles to some seconds.

First of all, the data is stored in the ring buffer to be read by the consumer task. The figure 7 shows the data transfert from the first producer of node 0 (variable name is Task1/producer1Value) to the last consumer of nodes 1 and 2 (variable name are Task2/node1Val and Task1/node1Val respectively). Each byte of the ring buffer has been tracked and set as user track in order to be displayed. The data stayed in the buffer between date 549876 and 647714, corresponding to 4891 μs in the time base. This takes most of the time in this data transfer. The figure 8 represents a zoom of the second phase of the transfer. As soon as the data is available in the task 3 of node 0 (variable name is Task3/consumerValue), it is sent to node 1 and 2, using data byte CAN message registers at address 0xef17. The transmission through the network is between date 648411 and 649741, corresponding to 68 μs. The CAN bus is configured at 1Mb/s, the frame contains 68 bits (standard frame with 2 bytes of data). At this date, the data is available for nodes 1 and 2 in the CAN register.

The node 1, that waits for the data using a polling method reads the data at date 650713. This takes a little time to get the variable because the process checks for new data in each CAN message objects (specific buffers that can store a message). The time elapsed to get the variable is 972 cycles, corresponding to 48μs.

The node 2, that waits for the data using the operating system interrupt service routine is slower, as the data is only read at date 651562 cycles (time elapsed : 1821 cycle, 91 μs). When an interrupt is thrown, the interrupt service routine of the Osek-Vdx is woken up which activates the task Task1. All this process takes some time and the interrupt method is nearly 2 times slower that the polling method, with this OS and that scenario. Of course, the polling method would be ineffective with other tasks in the same processor because it needs a lot of computation time.

4.2.2 Getting accesses at only one address

As each access to a tracked address is stored during simulation, indications about memory accesses can be obtained without following the generated path. For instance, getting all memory accesses on the first data byte of a CAN message object can give informations about the amount of messages received, the time elapsed between each message or the time elapsed from the availability of a message and the use by the processor (see figure 9). The same technique can be applied for example to the variable that stores the number of values in a buffer, that way, it is easy to tune the size of the buffer.

Tracking the program code is also possible. As the program code is read by the instruction pointer, when the entry point of a function is tracked, one can detect when the code is read. For instance, when tracking the first instruction of tasks, one can display when the tasks is woken up (see figure 10), in this case with tasks that can not be in a waiting state.

5 Conclusion and future work

This paper has presented a simulation platform based on SystemC, an Open Source industry standard HDL C++ framework. An accurate model of the C167CR processor which includes a CAN controller has been designed. A memory tracking method has been proposed. This method allows the analysis of distributed real-time applications in many ways:

- a tracking of program memory locations provides for instance a detection of task activation, and so allow the computation of timing caracteristics, for the used scenario,
- a tracking of data memory locations provides informations about buffer size, i/o operations and so on,
- a tracking of data movement provides a way to record the path followed by a data from its producer to its user.

The accuracy of the model is not discussed in this paper but it has been verified in comparison with the real execution of benchmarks on a C167 plateform.
consumer and can be used to give the end to end elapsed time.

This result shows that such a platform is suitable for software testing, debugging and analysis without being intrusive. The presented mechanism is the first foundation to debug and get information about the application under tests. Presently, we work on services for a higher level debugging and analysis:

**Task commutation:** the dates of occurrences of a task has been shown in a particular example. This will be extended in the general case (track of the stack pointer for a preemptive OS, cooperative OS, ...).

**stack size, buffer size:** the tracking method can be extended to give information of high level about the size needed for stacks/buffers,

**system calls:** the occurrences of system call must be directly integrated for a particular OS.

**time constraint violation:** violation of time constraints can be automatically detected.

---

**References**


