Abstract

This paper presents the ReTiS simulation tool and its use for real-time application analysis. Simulation concerns the operational architecture and takes into account the software (executable code of real-time tasks and real-time operating system) and the hardware (set of processors and networks). In this paper we show a mechanism that allows to detect task switch and to analyze task’s stack (usage and corruption). This mechanism neither requires any application nor operating system code modification. We show its interest for the analysis of the temporal behavior of real-time applications in a distributed system.

1 Introduction

The work presented in this paper takes place within the context of real-time embedded systems [8], [9]. These systems - when used for process control - have to react on stimuli emitted by the process, and the reactions have to respect the timing constraints of the process. Application timing constraints can be of various kinds and they can concern simple reaction (i.e. maximum alarm response time) or a complete chain of treatments (end to end timing constraints). So, the development of such systems requires the use of specific methods and techniques, particularly to prove the system meets 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 work presented hereafter is at the Operational Architecture level which is the result of the mapping of the Software Architecture (application functions) on the Technical Architecture (hardware and basic software as real-time kernel and communication system). 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 the execution support in a more detailed way, formal techniques are not appropriate any more. Nevertheless, with semi formal techniques one can do a validation taking into account the temporal behavior of the real-time executive and the communication system [12]. Lastly, when the executable code is available one can carry out a validation using a very accurate model of the hardware architecture. It is a step before the final test on the real target and our contribution is at this level. It concerns the analysis of the execution of the code (application code and basic software) on a very detailed model of the Hardware Architecture, using simulation. Our objectives are the temporal behavior analysis of the executable code in specific conditions in order to verify timing constraints, and to get time characteristics of the software. Interests of simulation techniques [3], [18], [1] at this step of the development - with regard to the final test on the target - are numerous and well known:

- one has no limitations of the final target: executable code in read-only memory, difficult control of task execution, difficult control of the environment behaviour;
- 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 behaviour;
- it is quite easy to set up observers for verifying that timing constraints are met, as well as to set up various time measurements.

Other real time system simulators use a higher abstraction level than ReTiS, such as scheduling simulators or RTOS simulators. Scheduling simulators ([11] for instance) use a model of the scheduling algorithm. A part of them considers the tasks as abstract time slots with probabilistic execution and/or arrival time. The other part sup-
port richer features like synchronization constraints for instance. Some scheduling simulators [10] [14] take into account distributed systems. RTOS simulators take into account the cost of the system calls and offer a finer model than scheduling simulators. However, these simulators do not model the hardware and do not use the actual code of the tasks. To our knowledge, there has been no work that gathers accurate distributed hardware simulation and high-level software analysis as it is done in ReTiS.

The paper presents some results on the development of ReTiS (Real-Time Simulator) [5], a simulator for Operational Architecture analysis. Figure 1 shows the architecture of the tool. Tool entries are the software to analyze (basic software and application software) as well as the scenarios for simulation. The tool uses a precise model of a hardware architecture (processors and networks). Tool outputs are a set of results corresponding to the input scenarios: track of variables, observation of task scheduling, dynamic analysis of stacks, code coverage, . . . The hardware architecture is based on the Infineon C167 processor and the CAN network but this study and the presented techniques do not rely on the given architecture and may be adapted to other processors and networks.

Due to the size of the paper we focus on only two aspects of this work (for information on variable tracking see [7]). The paper is organized as follows. First of all we present the modeling of a hardware architecture based on the Infineon C167 processor and its integrated CAN controller. This choice justifies itself since our applications are in the field of automotive embedded real-time systems. Then we present a mechanism allowing to determine the running task, assuming that each task running under the real-time OS control has its own stack. This mechanism is of primary importance to analyze the scheduling of the tasks. The mechanism does not require a stack-safe hypothesis. The second mechanism presented concerns the stack safety. This mechanism is based on a memory protection approach and aims at giving two important informations: the detection of stack corruption (overflow / underflow) and the real stack length used in the scenario. A static approach, as the one presented in [15] cannot deal with indirect branch calculated at runtime (function pointers or switch case control structure).

2 Modeling

2.1 Modeling the distributed architecture

All the analyses that are presented in this paper are not specific to a particular hardware architecture. The distributed architecture used as example is based on Infineon C167 processors connected through a CAN network. This kind of hardware architecture is commonly used in the field of automotive embedded real-time systems and fits our needs for a moderately complex system.

2.2 Working environment

The modeling and simulation platform environment is built on SystemC [19]. SystemC has been launched in september 1999 and is backed by a broad range of companies from semiconductors, systems, IP, embedded software and EDA industries. SystemC is an Open Source object-oriented framework written with C++ that enables Register Transfer Level (RTL) to system level design and simulation. A set of cooperating classes and templates permits the designing of 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) [17]. 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 the simulation engine to be customized if needed.

2.2.1 Modeling the Infineon C167

The main core of the CPU consists of a 4-stages instruction pipeline [20], 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 instruction behavior, independently of the time needed to execute the instruction, whereas a cycle accurate simulator takes into account the timings of the real system (it models pipeline and jump cache for instance). Both simulation schemes are interesting. A cycle accurate simulator is slow but essential for real-time systems simulation while 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 [6].

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 simulates one clock cycle. Additional hardware can still be easily connected through external devices.

2.2.2 Actions

A basic mechanism is needed to capture low level events of the simulation. We call this mechanism actions. An Action is a piece of code that is triggered when a specific event occurs. It is associated with each memory location and can be triggered on read, write or execution events. It is very useful in many cases such as:

- interfacing the CPU with other internal components,
- debugging purpose,
- other analyses such as variable tracking [7] and the one presented in sections 3 and 4.

The actions are independent of the processor considered, and thus the analysis presented in the next sections for task detection and stack usage analysis can be extended to other processor architectures (it is quite easy to analyze tasks and stacks with another processor, as soon as the actions are implemented in the simulator).

2.2.3 Modeling 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 is 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 for instance) and up to 8 bytes of data. There is an enhanced error control for critical applications.

In our case, the model accuracy must be fine enough to get precise timing information about the transmission duration of a frame and the date of emission. A simulator at the frame level would not be appropriate because of the bit stuffing behavior\(^1\). It implies that a frame must be evaluated at the bit level to compute its exact length.

2.2.4 Modeling the whole architecture

The whole architecture consists in as many nodes as needed. Each node is made of the SystemC modules (processor and CAN component), which are automatically synchronized. Memory accesses to the CAN Controller are modeled at a higher level than RTL (no modelling of the hardware control pin WR, RD, CS, \ldots) for performances reasons\(^2\). Anyway, the transactions are kept cycle accurate. The memory embedded in each node is included in the SystemC module of the processor as there is no shared memory. The whole architecture consists of as many nodes as needed communicating through the CAN network (figure 6). It is not necessary to use one clock for each processors/CAN network as the accuracy of the model is set on the CPU clock cycle. Only one clock is used for all processors and another one for the CAN network. It allows better computation time performances (there is only one SystemC event to take into account for all the processors, thus reducing the SystemC overhead).

3 Detecting the scheduling of the tasks

A mechanism that is implemented in the ReTiS tool is the detection of the scheduling of the tasks, in the context of a multi-tasking system.

---

\(^1\)To synchronize CAN nodes, an opposite bit is inserted in the frame when there are 5 consecutive bits of the same value. So, the frame length depends not only on the number of bytes transmitted, but also on the value of the transmitted data.

\(^2\)modelling at a higher level of abstraction requires less computation time
The stack may be accessed without the Stack Pointer. Using only a Stack Pointer analysis, the stack inter-
the compiler uses the instruction set correctly: a

However, stack corruption or specific usage of the stack
Stack Pointer is set in the stack area of the running task.
The current value of the stack pointer. In most cases the

In a multi-tasking system, each task has its own con-
text (stack and register set). So, it could be thought that
the detection of the currently running task is only tied to
the current value of the stack pointer. In most cases the
Stack Pointer is set in the stack area of the running task.
However, stack corruption or specific usage of the stack
by the compiler violate this rule. So, the value of the stack
pointer alone is not sufficient to detect the current task:

- the stack may be accessed without the Stack Pointer.
  For instance, on leaf functions (functions that does
  not call another function), the C compiler gcc uses
  the stack without changing the Stack Pointer. The
  Stack Pointer does not reflect the exact use of the
  stack. This is specially harmful for a stack analysis
  (see section 4);

- using only a Stack Pointer analysis, the stack inter-
  pretation can be infeasible. When stacks are contigu-
  ous, it is very difficult to detect if a stack overflows
  in the beginning of another stack as it can also be
  interpreted as a stack change.

Extracting high level information from low level infor-
information is often difficult. The technique presented in this
paper is based on following the execution path and detect-
stack modification resulting from a context switch at the end of functions and at the beginning of tasks. Few
hypotheses are required for this method:

- there is one stack for each task, the detection of a
  stack change can be interpreted as a task switch. No
  other knowledge of the internal OS behavior is
  required. The technique can be applied even if the
  source code of the OS is not available;

- the compiler uses the instruction set correctly: a
  function call must use the instruction that was de-
signed for it\(^3\). It is easy to adapt to a new architecture
by listing the instructions that are used for functions
calls.

The method does not require a Stack Safety hypothesis. As
it aims to be used in stack analysis (section 4), there is no
hypothesis on the validity of the stack pointer.

3.1 Guiding principle
Let us consider a scenario with 2 tasks (Task\(_1\) and
Task\(_2\)) as in figure 3. The three system functions wait-
Event, setEvent and terminateTask that appear in dark grey
are used and can perform a task switch. The two tasks are
in a ready state and Task\(_1\) has the highest priority. Task\(_1\)
starts (1) and calls the system function waitEvent (2).
This function performs a task switch to Task\(_2\) (3) be-
cause Task\(_1\) enters in a wait state. The function setEvent
is called, which makes Task\(_1\) in a ready state and call the
scheduler. A task switch is performed (5). Task\(_1\) is wo-
ken up and terminates through the function terminateTask
(6). Task\(_2\) resumes its execution (7). This scenario
is based on system functions from OSEK-VDX\(^4\), but the
technique presented in this paper is not specific to this op-

In this scenario, four tasks switches are done and must
be detected at four points (1, 3, 5 and 7) in the exec-
flow. The activation of both tasks (1 and 3) are detected by setting an action that is triggered on the exec-
tion of the first instruction of each task. The two other
points are more complicated. At 2, an action is set at the
next instruction that will be executed just after the return
from function setEvent; this action stores the stack pointer
value\(^5\). Then, at 5, the action is triggered. The current
stack pointer is compared with the one stored at 2. The
two stack pointers are equals if Task\(_1\) is the active one
(see section 3.2.2). The same method is applied to Task\(_2\)
with 4 and 7.

3.2 Implementation
The information that must be given to the simulator
is the task’s main function (the first function that will be
called the first time the task is run), the stack length and
the stack protection length (see section 4.2).

Actions are used to detect that instructions are exe-
cuted. When the instruction code is executed, the action
is triggered and a simulator function is executed to check
for a task switch. When an action is triggered, the sim-
ulation clock is stopped until the handling of the action
is done. Then the simulation clock is restarted and the sim-

\(^3\)For instance, the C167 uses the instructions calls, callr,
calli, calls, pcall, trap to call a function or a software in-
trupt. Other architectures have their own function call instructions like
bl and sc for the PowerPC.

\(^4\)OSEK/VDX OS is a multitask preemptive kernel [13] which is used
for embedded systems in the automotive industry.

\(^5\)because a function code can be shared by both tasks and more than
one action can be set at the same address. In this case, the saved value
of the stack pointer is used to detect the running task.
3.2.1 Static actions

Before the beginning of the simulation, the user has to declare the different tasks for the analysis to the simulator. Then, an action is created on the first instruction of the task function. This action is triggered on execution of the instruction and detects the code execution each time the task is started.

3.2.2 Dynamic actions

When a function call is executed (or an interrupt is handled), the stack value should be the same just before the execution flow branching ($SP_{call}$), and just after ($SP_{ret}$) as the Instruction Pointer (i.e. Program Counter) is stored on top of the stack. If the stack is changed in this period, there is a task switch.

An action is created and associated with the memory location that follows the function call or the next instruction to be executed in case of hardware interrupt.

An action is created at the address pointed by the instruction next to the function call. The action stores the stack pointer value ($SP_{call}$). For instance, for the following code snippet (the calls instruction represents a function call), an action is created during the execution of the instruction at address 0x28cc to be triggered at address 0x28d0. The action stores the value of the stack just before executing the calls instruction. This approach requires as many dynamic actions as there are nested functions.

```
0x28c8 : mov R9, #0
0x28cc : calls #0, global/GetResource
0x28d0 : mov R6, 0x8070
```

When the function or the interrupt handler returns (i.e. the instruction code at the address just after the call instruction or interruption is read), the current stack pointer and the one stored in the action are compared to detect the task switch. The action is then deleted.

There may be special cases when actions can be deleted; if an action is set, it checks that the stack pointer value is the deepest entry of all actions for this stack, and removes actions that do not follow this rule. It is very useful for stacks that are reset during execution. For instance, with our OSEK-VDX implementation the interrupt service routine uses a separate stack that is reset for each interrupt.

With this technique, the task switch is detected when the function that does it returns. The figure 4 shows a function $f$ that performs a context switch: just after the task switch and before the function returns, the simulator has a misunderstanding of the current task. This lack of accuracy is highlighted only in the few system calls that can switch tasks in the operating system. These OS system calls must be notified to the simulator just before starting the simulation. This way the simulator is able to detect when OS code is executed and do not generate any warning.

4 Stack analysis

A second high-level information that is extracted with the Retis tool is the stack analysis of the tasks. It is based on the task detection presented in the previous section. This information is used to check for stack safety (by detecting stack overflow/underflow) and to get the real use of stacks in the considered scenario. This last information is difficult to get and real time developers ask for this functionality to reduce the stack memory requirements.

The task’s stack analysis for real-time systems is widely based on the detection of tasks in the execution flow (see section 3), as this previous technique does not require a safe stack to be applied.

Our analysis is based on memory protection implemented in the simulator itself. For each stack defined in the system, actions that detect write access on the stack are added. Actions are added just before and after the stack range address to detect overflow/underflow.

Using an action placed at the first instruction of the task’s function that is triggered on the execution of the instruction, the initial stack value is obtained the first time the code is read. The action that detects the initial stack value has to check each time for the stack value because several tasks can share the same instruction code. Then,
the stack boundaries are detected with the stack length information that is given to the simulator.

### 4.1 Detecting the real stack use

In order to detect the real size used by stacks, an action that is triggered on a read access is placed on each byte of the stack address range with a flag. This flag is set on a write access. To get the real size used by the stack, the first byte that has been written during simulation is searched, starting from the end of the stack zone.

This method is not dependant on the way the compiler uses the stack. For instance, some C compilers do not modify the Stack Pointer in leaf functions (functions that do not call another function) or copy the stack into another register that is used for stack accesses. In such cases, the Stack Pointer does not always reflect the exact use of the stack.

### 4.2 Detecting stack overflow / underflow

The overflow / underflow detection is based on memory protection. Two protected zones are specified for each stack, just before and after the stack range. Each byte of these zones has an action that signals a stack corruption. These zones must be large enough to detect corruption, but not too large because they can interfere with memory that is used for other purposes.

With the relative lack of accuracy (function level accuracy) seen in section 3.2.2, the simulator has not to generate warnings that can occur because of a misunderstanding of the current running task, between a context switch and the end of the function. Obviously, stack overflows in such system calls would not be detected. This problem occurs only in functions that can switch the context.

### 5 Experimentation

The test bench is based on a 2 CAN nodes network. Each node is an Infineon C167 with an embedded CAN controller, that runs OSEK-VDX (see figure 6).

On node 0, the test program is a set of 3 tasks based on a producer/consumer model. The system is configured in full-preemptive scheduling mode. Both producers and the consumer are periodic tasks. The producers’ priorities are 5 and 6 and the consumer’s priority is 7 (0 is the lowest priority). The period of the consumer task is half of the period of the producer tasks in order to prevent a buffer overflow. The critical resource is a ring buffer with a priority set to 8 (rule of PCP protocol [16]). As soon as a data is read by the consumer, it is sent to node 1 through the CAN network.

Node 1 waits for a data from node 0 (interrupt triggered) and computes a result. As soon as a new CAN message is available, an interrupt activates the main task (Task1) that computes a result.

**Using this configuration, one can exhibit the scheduling of the tasks in a distributed system with task’s precedence between nodes and interrupt handling.**

### 6 Results

The test bench program presented in section 5 has been run on the simulator ReTiS for 2 millions of cycles (100 ms with a 20 MHz C167), requesting - for the whole architecture - a computation time of 3.81 seconds on a 2.8 GHz Pentium IV processor. During this time, 8 CAN frames are sent, which is sufficient for our example. The simulator outputs an execution trace consisting in the switching dates of the tasks as long as the stack usage. A graphical user interface which is part of ReTiS has been designed using the Cocoa / GNUStep framework. The graphical interface is needed to display the Gantt diagram of the tasks in a more efficient way than a text based approach. Time
in $x$ axis is in processor cycles since the beginning of simulation.

### 6.1 Gantt diagram of the tasks

Before the simulation, the tasks have to be defined, using symbolic names or addresses. In both nodes, each task $Task_i$ is defined with a stack of 64 bytes and no protected length (see section 4.2), as there is no stack corruption detection at this time. For each node, another task is used for initialization and is never called again. The implementation of the operating system uses 2 stacks. The first one is the main stack and is initialized just before calling the main function. The second one is used for the interrupt service routines (ISR2) and is initialized just before calling user interrupt functions. In this example, the ISR2 is not declared to the simulator. Undeclared tasks / ISR are all tied to an unidentified task. So for better results, it is important to declare all tasks / ISR at the beginning of simulation. The full configuration for node 0 is:

```plaintext
//stack for Task 1, size: 64 bytes
//memory protection: 5 bytes.
spy stack Task1 64 5
spy stack Task2 64 5
spy stack Task3 64 5
spy stack Task4 64 5
spy stack main 64 5 //stack of OS.
```

Figure 7 represents the Gantt diagram of the tasks. Two tasks may be running at the same time, but obviously only one for each node. The tasks in this example are very simple and interrupt calls take most of time. One can see that “as soon as” the consumer of node 0 ($task3$) returns, the first task of node 1 ($task1$) is woken up. The consumer of node 0 ends at date 1,035,693 cycles and the CAN frame is transmitted. The frame is received by node 1, and an interrupt is raised. This interrupt is taken into account in the Interrupt Service Routine, which has its own stack (The ISR is detected as a task, declared unknown, address is 0). At the end of the interrupt routine, the final consumer ($task1$) is activated. One can see that there is a delay of 2,477 cycles (between date 1,035,693 and 1,038,170, which represents 123.8 $\mu s$). This delay is due to CAN frame transmission time (including CAN component storing delay) and the interrupt handling of the operating system. The CAN transmission may begin before the end of $task3$ (the consumer of node 0).

This result has not taken into account functions or services that would not be available with a closed source operating system. For instance, no rules are based on internal behavior of the operating system, including context switch functions.

### 6.2 Stack analysis

When using the previous example, many warnings may appear about stack corruption, even if the stack range is

---

7The CAN frame is a standard data frame, with 2 bytes of data. It represents 60 bits and 4 extra-bits added by the bit stuffing for the frame transmitted at this date. The network is set to 1 Mbits/s, so the transmission time is 64 $\mu s$. 
sufficient for each task. This result is due to the fact that a task switch is not detected before the end of a function (see section 3.2.2). A warning with an internal function of the operating system may be generated by a lack of accuracy of our task analysis, but this is not a big problem for the stack analysis as only OS functions that manage stacks are concerned. In this way, system calls are specified to the simulator before the beginning of the simulation and do not generate any warning when simulating OS code. The functions to reject will be the same for each new application using the same OS.

If the application is run again, with a memory protection set to 5 bytes, no warning appears. The real stack use during the scenario is available. But in the case of a too large memory protection, many warnings may be generated. In our case, with a memory protection for Task1 set to 10 bytes, the memory protection interferes with registers mapped in memory.

7 Conclusion and future work

This paper has presented a simulation platform based on SystemC. An accurate model of the C167CR processor which includes a CAN controller has been designed.

Two methods to get high-level information from the low-level simulated behavior have been proposed:

A task switch detection. This method neither depends on the operating system nor the scheduling policy, including preemption or cooperation schemes. It takes the stack corruption into account and does not require an open source operating system.

Stack Safety analysis. This method is based on a memory protection approach and the task switch detection. It can detect stack overflow / underflow as well as the real stack length used in a scenario.

A graphical tool (part of ReTiS ) which displays the task’s Gantt diagram in a network context has been designed to show and quantify tasks’ precedences.

This result shows that such a simulation platform is suitable for software testing, debugging and analysis without being intrusive. Presently, we work on additional services for high-level debugging and analysis such as integrating the detection of the occurrences of system calls for a particular OS or detecting automatically timing constraints violations. ReTiS tool will be soon available under an Open Source licence.

References


8The 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 platform.