Model Extraction for ARINC 653 based Avionics Software*

Pedro de la Cámara, María del Mar Gallardo, and Pedro Merino[1]

University of Málaga
Campus de Teatinos s/n,
29071, Málaga, Spain
pedro.delacamara@gmail.com,{gallardo,pedro}@lcc.uma.es

Abstract. One of the most exciting and promising approaches to ensure the correctness of critical systems is *software model checking*, which considers real code, written with standard programming languages like C. One general technique to implement this approach is producing a reduced model of the software in order to employ existing and efficient tools, like SPIN. This paper presents the application of the technique to avionics software constructed on top of an application interface (API) compliant with the ARINC 653 specification (APEX), which is widely employed by the manufacturers in the avionics industry. The paper uses modelling techniques previously developed by the authors. However, these techniques are now extended to deal with new problems, like real-time aspects and scheduling. The paper also contains a novel testing method to ensure the correctness of the key part in the model: the execution engine that implements APEX services. This testing method uses SPIN to execute official test suites.

Keywords: Model extraction, software model checking, avionics, APEX, Real Time

1 Introduction

Application software for avionics, ranging from comfort and measurement software to critical flying control systems, are currently implemented on top of a shared network of processors following standard interfaces like ARINC 653 (Avionics Application Software Standard Interface) [1]. The applications share processors, memory and devices (sensors and input/output devices) and the whole system requires specific scheduling methods with real time features. So the verification of this kind of system is a real trend for model checker practitioners.

It is well known that one major problem of model checking for non-expert engineers is that the technique requires a deep understanding of both the modelling and the property languages supported by the tools. Furthermore, the manual construction process is susceptible to human errors due to misunderstandings or

* This work has been partially supported by the Spanish MEC under grant TIN2004-7943-C04

plain programming bugs. This is one reason to start many projects that can generate suitable models with minimal human interaction (see Feaver [9], JPF1 [7] and Bandera [4]). In [6] and [5], the authors develop a model extraction technique to deal with software built on top of well defined APIs. The approach was implemented for SPIN 4 (the same target as that of other related tools like FeaVer or Bandera). In this paper, we extend and apply our method to verify C applications running on top of APEX. The main extensions to our previous work consist in modelling time features and using conformance testing for checking the correctness of our model.

Modelling and verification of avionics software is also described in [10]. The problem of modelling real time in SPIN has been considered in [3]. Although, a detailed comparison with these works is given in Section 8, we may summarize the main contributions of this paper as follows:

1. The method to model *timing events* preserves the size of the state space into the limits suitable for verification with SPIN, like [3]. In addition, our approach can also benefit from abstract matching.
2. The approach in [10] is oriented to the verification of the operating system (OS), and not to the applications running on top of this OS. We model both the OS and the application and, in particular, we focus on debugging the application. Furthermore, our model of the ARINC execution engine is also available for traditional hand-made modelling of the applications.
3. In so far as the verification of the application depends on the correctness of the model of the OS, we automatically check the correctness of our ARINC model. To this end, we employ SPIN to do automatic testing, using the official test suites available in the avionics industry.

The paper is organized as follows. The preliminary material in Section 2 summarizes our general extraction approach to deal with software constructed on top of well-defined APIs [6]. Section 3 gives an overview of the APEX API and the partitioning scheme for avionics software. The main ideas of the model extraction approach are presented in Section 4 and the details of time modelling in Section 5. Some experimental results that confirm the feasibility of the method is given in Section 6. Section 7 shows the correctness of our approach through testing. Sections 8 and 9 conclude with a more detailed comparison with related works, and the conclusions, respectively.

2 Model extraction with Well-defined APIs

Existing approaches to verifying software by model extraction do not specifically address the problem of how to model services provided by the operating system. They are suitable for source code that only contains library functions that can be executed by the target model checker (see Feaver [7]). When the target model checker cannot directly execute all the operating system calls, it is necessary to add some extra hardness to complete the extracted models.

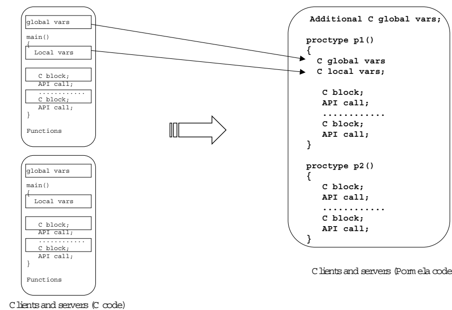

Fig. 1. Mapping scheme in SocketMC

In [6] we considered how to verify concurrent C applications that make extensive use of OS facilities through system calls. In our approach to model extraction, we construct a SPIN oriented model of the behavior of some operating system APIs. This model is used to automatically obtain a correct abstraction of the software that makes use of this API, for instance the Berkley-like Socket API. Following [9], we defined a mapping from the original C code to extended PROMELA. Tool SocketMC automatically transforms each API call into PROMELA. The new PROMELA model can be verified with standard SPIN.

As shown in Figure 1, mapping from the original code to PROMELA is done replacing every process (every `main` function) with a `proctype` definition. Then, the body of every proctype is filled using the extensions for C code (`c_decl`, `c_state`, `c_expr` and `c_code`). This is done breaking the C code at the points where a call to API appears. The final PROMELA code preserves the sequential execution of every C block code between two system calls. Thus, when verifying the model, SPIN interleaves blocks and system calls as atomic sentences.

By default, the PROMELA models produced by our model extractor contain all the C variables in the original code. Our method to produce *abstract matching functions*, presented in [5], allows us to automatically reduce the set of variables that should be actually managed to produce the state space. This optimization is also considered for models extracted from APEX based C applications.

This approach can be used for any kind of API, provided that the way of modelling the API calls preserve their semantics. So, in order to verify C applications written on top of APEX, we only need to correctly model the services provided by APEX and reuse our model extraction tool SocketMC. The definition of such API models is the aim of this paper.

3 The ARINC API for Avionics Software: APEX Interface

On-board avionics computing systems have evolved from federated specific computer systems towards generic to modular and integrated computers, like the

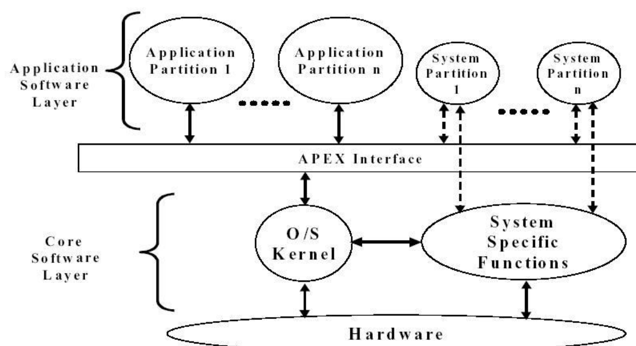

Fig. 2. APEX Interface

Integrated Modular Avionics (IMA). Most of the federated computers perform basically the same functions (input, processing, and output). A natural way of optimizing resources is to adopt a modular approach: to identify common parts, to standardize interfaces and to encapsulate services. Then, the next step is to share the hardware and software resources by integrating several functions on the same execution platform.

To enable multiple applications to be executed on the same computation resource, while avoiding error propagation, a robust isolation mechanism is used. Isolation is achieved by means of spatial and temporal partitioning, i.e., segregation of memory and time slots allocated to various application parts (or partitions) by means of software and hardware mechanisms.

The *Operating System* (OS) offers the basic and common services for all applications such as load, scheduling, and communication, through a well-defined API conforming to the ARINC 653 specification called APEX. We now briefly describe the main characteristics of APEX (see [1] for a more detailed description).

Partitioning.- One purpose of a core module in an IMA system is to support one or more avionics applications and to allow their independent execution. This can be correctly achieved if the system provides partitioning, i.e., a functional separation of the avionics applications, usually for fault containment (to prevent any partitioned function from causing a failure in another partitioned function) and for ease of verification, validation and certification. A partition is basically the same as a program in a single application environment: it comprises data, its own context, configuration attributes, etc. For large applications, the concept of multiple partitions relating to a single application is recognized.

APEX *Interface*.- As Figure 2 shows, interface APEX is located between the application software and the OS. It defines a set of facilities provided by the system for application software to control the scheduling, communication, and the status information of its internal processing elements. APEX also provides a com-

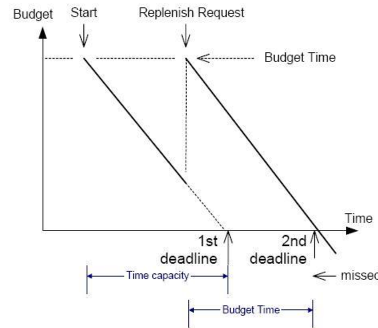

Fig. 3. Process Time Capacity

mon logical environment for the application software that enables independently-produced applications to execute together on the same hardware.

Scheduling.-Specification differences between *partition scheduling* and *process scheduling*. Scheduling of partitions is strictly deterministic over time. The *System Integrator* assigns one or more time windows to each partition. This is done in the fixed configuration within the core module. The scheduling algorithm runs repetitively with a fixed periodicity. Partitions have no priority by themselves. The scheduling unit is an APEX process. Each process has a priority. The scheduling algorithm is priority preemptive. During any process rescheduling event, the OS always selects the highest priority process in the ready state within the partition to receive processor resources. If several processes have the same current priority, the OS selects the oldest one.

Time Management.- As Figure 3 shows, a time capacity is associated with each process, and represents the response time given to the process for satisfying its processing requirements. When a process is started, its deadline is set to the value of the current time plus the time capacity. This deadline time may be postponed by means of service `REPLENISH`. This capacity is an absolute duration of time, not an execution time. This means that a deadline overrun will occur even when the process is not running inside or outside the partition window, but will be acted upon only inside a partition window of its own partition.

Interpartition and Intrapartition communication.- Interpartition communication is defined as the communication among two or more partitions executing either on the same module or on different modules. It may also mean communication between APEX partitions and external equipment. Interpartition communication is conducted via messages. Intrapartition communication mechanisms allow processes to communicate and synchronize with each other. All intrapartition message passing mechanisms must ensure atomic message access.

Service	Behaviour
GET_PROCESS_ID	provides a process identifier
GET_PROCESS_STATUS	returns the current status of the specified process. The current status of each process in a partition is available to all processes within that partition.
CREATE_PROCESS	creates a new process and returns its identifier. Partitions can create as many processes as the pre-allocated memory space supports. The consistency among process and partition parameters is checked. Assigning INFINITE_TIME_VALUE to PERIOD and TIME_CAPACITY defines an aperiodic process and a process without DEADLINE, respectively.
SET_PRIORITY	changes the current process' priority. The process is placed as the newest process with that priority in the ready state. Process rescheduling is performed after this service request only when the process whose priority is changed is in the ready or running state.
SUSPEND_SELF	suspends the execution of the current process, if it is aperiodic, until the RESUME service request is issued or the specified time-out value expires.
RESUME	resumes another previously suspended process. The resumed process will become ready if it is not waiting on a resource (delay, semaphore, period, event, message). A periodic process cannot be suspended, so it cannot be resumed.
STOP	makes a process ineligible for processor resources until another process issues START.
START	initializes all attributes of a process to their default values, and resets the runtime stack of the process. If the partition is in NORMAL mode, the process' deadline expiration time and the next release point are calculated.
GET_MY_ID	returns the identifier of the current process.
GET_PARTITION_STATUS	provides the status of the current partition.
SET_PARTITION_MODE	sets the operating mode of the current partition to normal after initialization of the partition is completed. The service is also used to set the partition back to idle (partition shutdown), and to cold start or warm start (partition restart), when a serious fault is detected and processed.
TIMED_WAIT	suspends execution of the requesting process for a minimum amount of elapsed time. Value zero allows the round robin scheduling of processes with the same priority.
PERIODIC_WAIT	suspends the execution of the requesting process until the next release point in the processor time line that corresponds to the period of the process.
GET_TIME	returns the value of the system clock. The system clock is the value of a clock common to all processors in the module.
REPLENISH	updates the deadline of the requesting process with a specified BUDGET_TIME value. It is not allowed to postpone a periodic process deadline past its next release point.

Table 1: Modelled services

4 Modelling Processes

Following the approach described in [6], our PROMELA model is composed of several application processes, extracted from the application C source code, and an environment that closes the model. During the extraction, the API calls in the original source code are translated into calls to the environment. Consequently, the environment is the process that provides the APEX services to the application processes, and that stores all the state information needed as global data.

We have not modelled every APEX service or functionality. As a first step, we have focussed on what, in our opinion, is the most critical aspect to be modelled in SPIN, that is, the APEX Time Management. Table 1 above shows services modelled with a brief description of their behaviour.

Since the present paper focuses on modelling APEX Time Management, and several APEX features and services are not implemented in this first phase, the following limitations have been imposed: only one partition is allowed, processes cannot be restarted after being stopped, partitions cannot be restarted, error handler, and Health Monitoring and recovering actions are not supported.

4.1 Modelling the Process Scheduling

Applications running on APEX are composed of processes. Processes are scheduled according to the following APEX process scheduling rules: 1) the scheduling unit is an APEX process, 2) each process has a priority, 3) the scheduling algorithm is priority preemptive, 4) during any process rescheduling event, the OS always selects the highest priority process in the ready state, and 5) if several processes have the same current priority, the OS selects the oldest one.

In our model, scheduling is included in the environment and ensures that, in every state, only one PROMELA process is executable by SPIN. Scheduling also ensures that the APEX scheduling rules are followed when selecting a process for processor resources.

Modelling preemption without state explosion. The first issue of modelling APEX scheduling comes from the *process preemption* requirement. APEX specification states that a process may be preempted whenever a re-scheduling event takes place (for instance, when a higher-priority process is resumed after a timeout expiration). Implementing this requirement increases the complexity of the resulting model, which may lead to state explosion during verification.

To solve this problem, we divide the source code into code blocks and APEX API calls, as introduced in [6] (see Figure 1). Code blocks operate only at local scope (i.e. read and write local variables). API calls operate at global scope (i.e. read or write global variables). Notice that application processes can only communicate with other processes or with the environment through API calls.

¿From the model checking point of view, code blocks perform non-visible local actions, that neither affect other processes nor the environment. Thus, when a process is executing a code block, it does not matter in which code point it is preempted. Therefore we may merge all the sentences of a code block in an *atomic block*, allowing processes to be preemptable only before or after the execution of a code block. The idea of merging local sentences is well known, and it has already used in [6].

API calls behave differently. They use global information and have global effects but, since they are already executed atomically in the real system, it is also correct setting them as atomic in the model.

Controlling process execution. The second important issue when modelling the APEX scheduling is to ensure that the scheduling rules are fulfilled. Scheduling must be able to stop the execution of a process and resume another one. To do this, we use a `provided()` sentence in each PROMELA `proctype` declaration.

Sentence `provided()` appends an executability clause to every transition of that process, disabling its execution when it is false. For example, global variable `_curSchProc` in clause "`proctype p1() provided (_curSchProc == _pid)`" stores the pid of the process currently in execution. Thus, sentence `provided()` is only true for the process whose `_pid` matches `_curSchProc`. Thus, the scheduler can control the execution of processes by modifying the `Pid` stored in the variable.

Storing process attributes. Scheduler must keep information about the attributes of each process to correctly realize the context switch. These data are stored as C-structures in the environment, using the embedded-C primitives provided by SPIN 4. Attributes may be static, if their value does not change, or dynamic. Process attributes are shown in Table 2 below.

Due to optimization reasons, only dynamic attributes are included in the SPIN state-vector. Static attributes are marked as `Tracked UnMatched` data (see references [6], [5] and [8]). Hiding these data during the comparison of states (`Matching`) does not discard any significant behaviour, since static attributes are written only once, during the partition initialization phase, when the process is created and the initialization phase is completely deterministic.

Static Attributes

Name	
Base priority	Initial priority
Period	
Time capacity	the amount of time in which the process must finish its work (it is used to calculate the process deadline)
type of deadline	it can be `HARD` or `SOFT`

Dynamic Attributes

Current priority	
Deadline	Absolute time when time budged will expire.
State	APEX state of the process: `WAITING, RUNNING, DORMANT, READY`.
Waiting cause	Reason why a process is `WAITING` (e.g. for a resource).
Position in queue	If the process is `WAITING` or `READY`, position in the queue of processes waiting for the same cause. This attribute is used when there are several candidates to be awaked/run and the scheduler must choose the oldest one.
Resource	If the process is waiting for a resource, this is its identifier (e.g. a communication port).
Timeout	If the process was suspended with timeout, absolute time when the process will be resumed.
Time wait	If the process called `TIMED WAIT`, absolute time when the process will continue its execution.

Table 2. Process attributes

Calling the scheduling functionality. Scheduler is called when the so-called re-scheduling events occur. Re-scheduling events are triggered when a change in the environment may modify the executability of one process. Examples of these events are API calls (e.g. `SET_PROCESS_ PRIORITY` and `SUSPEND_SELF`), *time*

events (e.g. deadline or timeout expiration), the reception of messages in ports, and intra-partition communication (events up, signaling of semaphores, etc.)

5 Modelling time

The APEX time management model is based on the notion of *time events*. In respect to model checking with SPIN, a *time event* is a global event, triggered by the environment, and associated to a point in the timeline.

The *time event* concept is related to the classic "tick" signal used in other real-time models. A "tick" signal usually indicates that certain amount of time has elapsed. It is used to interrupt the execution of the model, and to allow the increment or decrement of a time counter. In some cases, increasing/decreasing is not the only action. Other actions, such as awakening a process or notifying a deadline violation, are also carried out as a response to this "tick". In contrast to these signals, *time events* are *only* triggered *when* other actions must be taken. In other words, a *time event* is a special "tick" signal that is only triggered if its effect on the model goes beyond increasing/decreasing a time counter.

Our time management model includes a system clock that keeps track of the time flux, and that is updated by *time events*. That means that the system clock is only updated when a time-related event takes place. If a monitor process observed the system clock during an execution sequence, it would notice that the clock keeps the same value for long time and then, after a *time event* takes place, it jumps to a new value. The most important issue is that the size of the jump is completely dependant on the *time event*, and on the behavior of model itself. It is worth noting that, in terms of model checking, this approach causes a state-space reduction by avoiding "tick" signals that interrupt the model only to update time counters.

5.1 Life-cycle of *time events*

Time events may be armed statically, before starting verification, or dynamically by the environment during execution of the model. The typical life cycle of a *time event* is as follows:

- An application process calls an APEX service (e.g. `TIMED_WAIT(500ms)`).
- The environment (i.e. the APEX OS) takes control and suspends the calling process. Then it arms a `Waiting_Timeout` *time event* that will be triggered `500ms` in the future.
- The environment returns the execution to another ready application process.
- Eventually (i.e. `500ms` in the future), the environment will trigger the *time event*. Then execution of the running process will be interrupted, so that the environment can awake the suspended process.
- Finally, the environment updates the system clock in `500ms`, removes the *time event*, and returns execution to the appropriate application process.

5.2 Implementing and using *time events*.

Regarding the implementation details in SPIN, the environment keeps a list with all the armed *time events* as a global state variable. The impact of the list in the state-vector size is not remarkable, since it is optimized for memory saving.

Time events may be used in different ways, depending on how they are armed and triggered. In this project, we have explored two ways of modelling APEX time management in SPIN. They are called *Time Management Types* and each one corresponds to a different modelling requirement. *Time Management Type 1* (TMT1) is used for modelling applications when the execution times of the application code are unknown. *Time Management Type 2* (TMT2) is a refinement of TMT1 taking these execution times into account.

Time Management Type 1: Execution times are not used. In the early stages of software development for industrial processes, execution times are often not available. However, in these first phases, verification by model checking can be very useful as a mean of discarding design errors. Even if execution times are unknown, the piece of software under analysis may already include in the code some time values (inherited from the requirement or design phases). TMT1 provides a model of the environment capable to interpret these time values, but without using specific execution times.

TMT1 makes an over-approximation of the model assuming that the execution of each application code block may take a variable amount of time ranging from 0 to infinite time units, but only at those time points where a *time event* is triggered. Evidently, this over-approximation produces execution traces that will never happen in real execution.

This over-approximation is implemented by the process `TimeEventTrigger`, whose code appears below. This process is in charge of triggering *time events* and runs parallel to the application processes.

```
proctype TimeEventTrigger () {
      do                                  \
      :: (1) ->c_code {ma_Tempus_Fugit();} \
      od;
}
```

Process `TimeEventTrigger` contains an infinite loop with a call to the environment function `ma_Tempus_Fugit` that triggers *time events*, and updates the system clock. Since `TimeEventTrigger` may always be executed, every transition of the model may non-deterministically choose to execute either an atomic application code block (whenever there exists at least one enabled process), or the `TimeEventTrigger` sentence `Tempus_Fugit`.

We now describe the possible execution sequences produced by SPIN when the application code blocks and process `TimeEventTrigger` are interleaved. In the discussion, we denote with $A_0, A_1, \ldots, A_k$ the application code blocks of the model under analysis, $et(A_i)(0 \leq i \leq k)$ being their respective execution times. Similarly, we assume that $T_0, T_1, \cdots$ are the sucessive `TimeEventTrigger` transitions, and that $tp(T_i)$ is the time point associated to the *time event* triggered

in T_i. We use $\rightarrow$ to represent the order in which transitions are executed in a given sequence of transitions. It is important to remember that the system clock is only updated during the *time event* triggering which, in this case, means that it is only updated by `Tempus_Fugit`.

Starting with one `TimeEventTrigger` transition T_0, SPIN explores all the following execution sequences:

- $T_0 \rightarrow A_0 \rightarrow A_1 \rightarrow \cdots$, that is, no `TimeEventTrigger` transition takes place. Then, the system clock remains unchanged, and therefore, $et(A_0) + et(A_1) + \cdots = 0$.
- $T_0 \rightarrow A_0 \rightarrow \cdots A_n \rightarrow T_1 \rightarrow \cdots$, that is some application code blocks are executed between T_0 and T_1. After T_1 is triggered, the system clock is updated, and therefore, $et(A_0) + \cdots + et(A_n) = tp(T_1) - tp(T_0)$.
- $T_0 \rightarrow T_1 \rightarrow A_0 \rightarrow \cdots$, that is, the first block A_0 is executed after T_1. This sequence models the behavior where $et(A_0) > tp(T_1) - tp(T_0)$. It can also represent a behavior where execution of A_0 is disabled until the *time event* T_1 is triggered.
- $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots$, that is, no application code block is executed. This sequence models the behavior where $et(A_0)$ is infinite. It can also represent a behavior where execution of A_0 is indefinitely disabled.

TMT1 can be used to verify properties related to the ordering of execution of application code blocks (A_i) and `TimeEventTrigger` transitions (T_i). This includes all non-real-time temporal properties usually checked in SPIN. In addition, TMT1 can only deal with *some* of the properties involving real time values. For example, the existential property *"Process P1 is able to do its work before its deadline"* can be checked. SPIN will try to find an execution in which P_1 *is able* to finish successfully. However, property *"Process P1 shall always do its work before its deadline"* cannot be proven because it will always be violated by the execution sequence where one code block has an infinite execution time.

Time Management Type 2: Execution Times are used. The *Time Management Type 2* refines TMT1 by adding information about the *execution times* of the application processes. TMT2 assigns a fixed execution time to each code block. These times may be real data, estimations or even purely hypothetical values.

In a real system, a *time event* may happen at any time as, for instance, during the execution of a code block. However, code blocks are modelled as atomic sentences, and therefore, their execution cannot be interrupted. But this is not a problem, since, by definition, code blocks only perform local computation, with no visible effects on other processes, and it does not matter whether a *time event* is triggered during or just before a code block, as it does not influence the whole behavior of the model.

The strategy of TMT2 is to delay the execution of a code block until it can be executed without being interrupted by *time events*. Whenever a code block is delayed, its remaining execution time is decreased in an amount equal to the

delay. This is because the model considers that, in the real system, the code block would be *partially* executed. Eventually, when the remaining execution time is so short that no *time event* can interrupt the code block, the actual code block will be executed.

A special *time control logic*, which is considered part of the environment, is added at the beginning of each code block. Both the whole code block and time control logic are included in an atomic sentence. The following example illustrates the way time control logic works.

```
1   CP1:
2   Atomic { /* Init of Time Control Logic of Block 1 */
3   if RemainingExecTime(_pid) = 0 then
5       RemainingExecTime(_pid) = CP1_EXEC_TIME;
6   end if
6   if ( (T_NextTimeEvent - CurrentTime) < RemainingExecTime(_pid) then
/* Partial code block execution */
7       RemainingExecTime(_pid) = RemainingExecTime(_pid) - (T_NextTimeEvent  - CurrentTime);
8       CurrentTime = T_NextTimeEvent;
9       Trigger Time Event;
10      goto CP1;
11  else /* Complete code block execution */
8       CurrentTime = CurrentTime  + RemainingExecTime(_pid);
9       RemainingExecTime(_pid) = 0;
11  end if
 /* End of Time Control Logic of Block 1 */
     ///////////////////////////////////////////////////////
              APPLICATION CODE BLOCK
    ///////////////////////////////////////////////////////
12  }
```

Firstly, the time control logic stores the fixed execution time for the code block into the global variable `RemainingExecTime`, which is specifically assigned to the corresponding process. This variable is only updated when its value is zero.

Then, the time to the next armed *time event* is calculated, and if it is possible to completely execute the code block before the next *time event* is triggered, variable `RemainingExecTime` is set back to zero, and the system clock is increased in the same amount. In other words, the elapsed time is equal to `RemainingExecTime`. After setting the system clock, the code block is executed as usual.

Otherwise, if the block code cannot be completely executed, the difference between the next *time event* and the current system clock time is calculated. This value, that represents the time elapsed during which the code block has been partially executed, is subtracted from `RemainingExecTime`. Just after that, the *time event* is triggered, and the process jumps back outside the atomic block. Notice that, as a result of the triggered *time event*, the process may be disabled. Eventually, when the process runs again, it will try to execute the same code block, performing a new iteration of the time control logic. Since this time variable `RemainingExecTime` is not zero, its value is not updated. This iteration goes on until the code block can be completely executed.

This implementation also includes process `TimeEventTrigger`, whose code appears below, and that uses the `timeout` PROMELA sentence to trigger *time events* when the rest of processes are blocked.

```
proctype TimeEventTrigger () {
  do                                        \
  :: (timeout) ->c_code {ma_Tempus_Fugit();}  \
  od;
}
```

In summary, if an *execution event* is a fictitious event that occurs when a code block finalizes its execution, it is possible to state that TMT2 preserves the ordering of *time events* and *execution events*.

However, a question still remains. What happens with the API service calls which are not completely local? The answer is that in a real system, a *time event* cannot interrupt API calls. The concrete effects of a *time event* (for example, a DEADLINE) triggered during an API call are mostly implementation dependant. Usually, the effects can be seen after the process returns from the call. In our model, API calls are delayed until all the interrupting *time events* are triggered.

6 Experimental results

We now present an example to evaluate our APEX environment. The application code has been instrumented in such a way that SPIN generates one execution trace for each possible execution sequence. By analyzing the traces, we may understand the order in which sentences were executed during verification, and the time when it happened (measured with the modelled system clock). We have used *Time Management Type 1*.

The example consists of one process P1 with the following behaviour. Process P1 obtains its own identifier (GET_MY_ID), modifies its own priority (SET_PRIORITY) four times and enters the waiting state (TIMED_WAIT) for 500 time units. After awaking, it reads the system clock (GET_TIME) and stops itself (STOP_SELF). The complete code of the example may be found at Appendix A.

Process init, whose code appears below, creates P1 (CREATE_PROCESS), starts it (START) and begins the normal process scheduling by setting the partition in NORMAL mode (SET_PARTITION_MODE). Since P1 has a time capacity of 5000 time units, its deadline shall be set 5000 time units. For this example, we modelled a minimal Health Monitoring functionality that stops process P1 when its deadline is reached.

```
init{ atomic {
    c_code {
        MODEL_A653_Init();   };
    c_code{
        strcpy (a_att.NAME, "p1");
        a_att.STACK_SIZE = 200;
        a_att.BASE_PRIORITY = 30;
        a_att.PERIOD = INFINITE_TIME_VALUE;
        a_att.TIME_CAPACITY= 5000;
        a_att.DEADLINE = SOFT;
    };
        A653_CREATE_PROCESS(&a_att, &a_proc,
                                        &a_return,p1);
        A653_START(a_proc,&a_return);
        A653_SET_PARTITION_MODE(NORMAL,&a_return);
        run clock();
      };
      };
```

As described in Section 5.2, Time Management Type 1 makes use of process TimeEventTrigger to trigger *time events*. In order to explore every possible sequence of *time events* and application code-blocks, Time Management Type 1

takes advantage of the non-deterministic interleaving between this process and the application processes. In particular, the instrumentation code added to this example generates 10 traces, one for each possible execution sequence. We now discuss them in detail.

Trace 1

Time	Process Executed	Description
0	`TimeEventTrigger`	Next *time event* t = 5000
5000		`DEADLINE`

In this trace, `TimeEventTrigger` runs before `P1` and triggers the `P1-DEADLINE` *time event*. In consequence, the execution is stopped before `P1` can run. In a real system, this behavior takes place if the execution time of the first code-block of `P1` takes more than 5000 time units.

Traces 2-6

The second trace corresponds to the scenario where `P1` is able to execute sentence (`GET_MY_ID`) before its deadline is triggered. Note that after the execution of `GET_MY_ID`, the system time has not advanced.

Time	Process Executed	Description
0	`P1`	CP11: `GET_MY_ID`
0	`TimeEventTrigger`	Next *time event* t = 5000
5000		`DEADLINE`

The next four traces are similar to the previous one. In each of then, `P1` is able to execute one additional sentence `SET_PRIORITY` before the deadline is triggered.

Traces 7-10

In the seventh trace, `P1` enters a waiting state for 500 time units. A *time event* is armed at $t = 500$ and, since no other process is executing, it is triggered. Finally, the deadline of `P1` is triggered.

Time	Process Executed	Description
0	`P1`	CP11: `GET_MY_ID`
0	`P1`	CP12: `SET_PRIORITY`
0	`P1`	CP13: `SET_PRIORITY`
0	`P1`	CP14: `SET_PRIORITY`
0	`P1`	CP15: `SET_PRIORITY`
0	`P1`	CP15: `TIMED_WAIT`
0	`TimeEventTrigger`	Next *time event* t = 500
500		Awake `P1`
500	`TimeEventTrigger`	Next *time event* t = 5000
5000		`DEADLINE`

The last three traces show how `P1` continues its execution after `TIMED_WAIT`. The last trace represents the behavior where the execution of `P1` is so fast that its deadline is not triggered.

SPIN search reached a depth of 30, with a total state-vector size of 228 bytes. From these 228 bytes, 107 bytes are used to store the environment state and the rest are for application or instrumentation variables. The example confirms

that Time Management Type 1 can be used as an over-approximation to analyze applications when there is no information about execution times.

7 Testing the model of the API

In a real system, application processes use the OS services described in the specification ARINC 653 Part 1 [1]. In the verification model, applications call services provided by the verification environment. To ensure the correctness of the verification, the behavior of the environment services must match the real OS services. As a first step to establish the soundness of the model proposed here, we have carried out an exhaustive testing campaign. The procedure consists in running a battery of *Test Cases*, written in PROMELA+C code, which call every implemented service in every possible condition. Each Test Case provides a fail/pass verdict, depending on the behavior of the services called.

Test Cases are defined considering the ARINC 653 Part 3 "Conformity Test Specification" document [2]. This specification gives a description in natural language of an APEX conformity test-battery. In other words, this battery checks if the APEX services provided by an OS are conformed to the ARINC 653 Part 1 specification. In consequence, it can also be used to check if the services implemented in our environment are also conformant.

Test Cases are classified into functional and robustness tests. Functional tests check that the service works in normal use conditions. Robustness tests check how the service works in abnormal conditions (e.g. it returns the appropriate error codes). The result is a complete conformance test suite. Our work firstly consisted in selecting the test definitions applicable to the current services implemented in the environment. Then we built the PROMELA-C Test Cases and finally we checked that no Test Case returned a fail verdict. The execution of the test cases and the PROMELA model of APEX is done with the validation facilities of SPIN. Due to space restrictions, we cannot describe in detail the codes for Test Cases developed. As an example, Appendix B shows the Test Case checking the `SET_PRIORITY` service when the `PROCESS_ID` input parameter is not valid.

It is worth noting that during testing we have used *Time Management Type 2* because we are assuming a deterministic evolution of time.

8 Related work

Since this paper focuses mainly on modelling the time management aspects of APEX, in this section, we have concentrated on the previous references which have modelled real-time in SPIN.

The proposal presented in [10] shows how to verify a microkernel used in the avionics industry, by constructing a PROMELA model of the Honeywell DEOS operating system. In order to close the OS model, all the possible interactions between the DEOS model and the external world are modelled inside the environment. Basically, this environment includes threads requiring services from the OS and time-related interruption sources. Our work has the opposite goal.

We aim at verifying avionics applications which access OS services through the APEX interface. In our case, we semi-automatically extract PROMELA models from applications source code (see [6]). In order to close the application model, we need to use an environment able to provide APEX services to the applications.

In the DEOS environment, one message is periodically generated to indicate that a higher priority thread may become schedulable. After receiving this message, the kernel checks if the current running process must be pre-empted by the higher priority thread. In our case, since the environment is the OS, and the threads are the applications to be verified, whenever a higher priority process becomes schedulable, the environment, that is the OS, disables the execution of the low-priority processes and enables the high-priority one.

The DEOS environment has two time-related interruption sources. The first one periodically interrupts the kernel, in order to check if a higher priority has become schedulable. The second one interrupts the kernel whenever the running process exhausts its time budget. The DEOS environment combines both interruptions in one process, in order to coordinate them and avoid "impossible" behaviours. Similarly, our environment includes a `TimeEventGenerator` process that may be seen as the combination of every time-related interruption source applicable on APEX (waiting timeout expiration, time budget expiration, etc.) However, instead of ticks, our `TimeEventGenerator` triggers *time events*. Each *time events* has an associated *time point*. The environment is able to know the current system time, just by reading the time point value of the last *time event* triggered. This feature allows our environment to provide the system time to the applications. Since both the environment and the applications are aware of system time, it is possible to use timing values in the properties to be verified (i.e. in `LTL` formulae).

The work presented in [3] is a time extension for PROMELA. The principles explained in that work are generic and may be applicable to different modelling problems. In our case, we aim at modelling the time management as described in the APEX specification. The context of our approach is automatic model extraction of avionics applications and APEX environment modelling. That means that we are in the position of using modelling techniques specially tailored for this context. On the other hand, some of the techniques and assumptions made may not be applicable to other modelling problems.

In the discrete time model of [3], time is divided into slices. The actions take place inside these slices, making it possible to obtain a measure for the time elapsed between events belonging to different slices. However, within a slice, only the relative ordering between events is known. The time elapsed between events is measured in ticks of a global digital clock that is increased by one with every such tick. In our case, a time slice may be considered as the time elapsed between two consecutive *time events*. The main difference is that the time slice size is always variable and depends on model behavior. Furthermore, *time events* may only happen between two atomic blocks of application codes. Another significant difference is that we have identified two different use scenarios, depending on whether the execution time of application code blocks is known

or not. If execution time is not known, the environment assumes the worst case over-approximation, that is, the execution time of each code block may be any value from 0 to infinite. In practice, this means that between two *time events*, a process can execute a non-deterministic number of code blocks, unless it makes an API call involving waiting for an event (`TIMED_WAIT`, etc.).

In principle, as in [3], only the relative ordering between events is known within a slice. However, if the execution time of code blocks is known, the environment making it possible to know the absolute time in which a block of code was executed. One important point of the work [3] is that it is compatible with SPIN partial order reduction algorithm (POR). This is not our case, since we use the PROMELA the `provide` sentence which is incompatible with POR. However, this is not a big performance issue for our model, since the main objective of POR is to reduce the state explosion caused by non-deterministic process interleaving. Due to the way in which APEX processes are scheduled, this kind of state explosion rarely appears during verification.

Another related development is the RT-Spin package of [11]. RT-Spin extends PROMELA in order to deal with Real Time. This is the main difference with our work, since we rely on standard PROMELA.

Finally, another main difference with the works referred to above is that most of the environment is implemented in C code. This is done using the embedded C capabilities of SPIN 4. The embedded C code allows us to use data abstraction techniques, as those explained in [5].

9 Conclusions

The first and most important conclusion of this work is that verification of APEX-based avionics applications is feasible with SPIN. We have proven that SPIN is able to model APEX-like Real-Time management in a correct and efficient manner, if the right methods and assumptions are used. Our past experience in modelling APIs ([6], [5]). tells us that other features of APEX (e.g. inter-partition communication, process synchronization, etc) can also be modelled in SPIN. Actually, we are extending SOCKETMC to obtain a more generic model extractor that allows us to automatically verify APEX based applications.

The second conclusion is that verification methods and assumptions must be adapted to each use-scenario. In our case, if application execution times are not known before the verification, then we must use methods and assumptions that cope with this incertitude (i.e. Time Management Type 1). On the other hand, when execution times become available, we must refine the verification using more accurate methods and assumptions (i.e. Time Management Type 2).

In respect to future work, we have several parallel lines of study. First, we plan to expand the set of modelled APEX services. The next step will be to include inter-partition and intra-partition communication services. These new services will be integrated in the ongoing extension of SOCKETMC[6].

We also want to improve the approach by using memory optimization methods based on data abstraction [5]. We have detected that in some use-scenarios,

the execution times are known, but with some degree of incertitude. For these scenarios we want to build a Hybrid Time Management, where execution times are non-deterministically chosen among a limited set of values.

Finally, we realize that APEX static configuration (e.g. ports, partitions, time-schedule, etc.) must also be incorporated into the verification model. For this purpose, we plan to enable our environment to read external static configuration information. In the long term, we want to be able to parse APEX configuration XML files and translate them into a SPIN-friendly configuration.

References

1. ARINC. *ARINC Specification 653-2: Avionics Application Software Standard Interface Part 1 - Required Services.* Aeronautical Radio INC, Maryland, USA, 2005.
2. ARINC. *ARINC Specification 653-2: Avionics Application Software Standard Interface Part 3 - Conformity Test Specification.* Aeronautical Radio INC, Maryland, USA, 2006.
3. D. Bosnacki and D. Dams. Integrating Real Time into SPIN: A Prototype Implementation. In *Proc. of the FIP TC6 WG6.1 Joint Int. Conf. FORTE XI / PSTV XVIII '98*, pages 423–438, 1998. Kluwer, B.V.
4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE '00: Proc.of the 22nd Int. Conf. on Software Engineering*, pages 439–448, 2000. ACM Press.
5. P. de la Cámara, M. M. Gallardo, and P. Merino. Abstract matching for software model checking. In *13th Int. Workshop on Model Checking of Software (SPIN06)*, pages 182–200, 2006. Springer-Verlag.
6. P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined apis: the socket case. In *FMICS '05: Proc. of the 10th Int. Workshop on Formal methods for Industrial Critical Systems*, pages 17–26, 2005. ACM Press.
7. K. Havelund and T. Pressburger. Model Checking Java Programs using Java Pathfinder. *International Journal of Software Tools for Technology Transfer*, 2(4):366–381, 2000.
8. G. J. Holzmann and R. Joshi. Model-driven Software Verification. In *11th Int. Workshop on Model Checking of Software (SPIN04)*, pages 76–91, 2004.
9. G. J. Holzmann and M. H. Smith. Software model checking: Extracting Verification Models from Source Code. *Software Testing, Verification & Reliability*, 11(2):65–79, 2001.
10. John Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *ICSE '00: Proceedings of the 22nd Int. Conf. on Software Engineering*, pages 488–497, 2000. ACM Press.
11. S. Tripakis and C. Courcoubetis. Extending Promela and Spin for Real Time. In *Proc. of the 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS96)* , pages 329–348, 1996. Springer-Verlag.

A Code of the example

```
c_state "PROCESS_ID_TYPE apid" "Local p1"
c_state "RETURN_CODE_TYPE ret" "Local p1"
c_state "SYSTEM_TIME_TYPE time" "Local p1"
proctype  p1 () provided ( _curSchProc == _pid )
{ CP11:
    atomic {
        TRACE(11);
        A653_GET_MY_ID(&(Pp1->apid),&(Pp1->ret),Pp1->_pid);
    }
CP12:
    atomic {
        TRACE(12);
        A653_SET_PRIORITY(Pp1->apid,34,&(Pp1->ret));
    }
CP13:
    atomic {
        TRACE(13);
        A653_SET_PRIORITY(Pp1->apid,34,&(Pp1->ret));
    }
CP14:
    atomic {
        TRACE(14);
        A653_SET_PRIORITY(Pp1->apid,34,&(Pp1->ret));
    }
CP15:
    atomic {
        TRACE(15);
        A653_SET_PRIORITY(Pp1->apid,34,&(Pp1->ret));
    }
CP16:
    atomic {
        TRACE(16);
        A653_TIMED_WAIT(500,&(Pp1->ret),Pp1->_pid);
    }
CP17:
    atomic {
        TRACE(17);
        A653_GET_TIME(&(Pp1->time),&(Pp1->ret));
        c_code{
            printf("Time = %u \n",Pp1->time);
            };
        }
CP18:
    atomic {
        TRACE(18);
        A653_STOP_SELF(Pp1->_pid);
    }
} }
```

B Test Case checking SET_PRIORITY

```
//////////////////////////////////////////////
//////P1
//////////////////////////////////////////////


proctype P1 () provided ( _curSchProc == _pid ) {
    ...
}


//////////////////////////////////////////////
//////Master Test
//////////////////////////////////////////////


c_state "RETURN_CODE_TYPE ret" "Local Master_Test"


proctype Master_Test () provided ( _curSchProc == _pid ) {


// Invalid Process Id
    atomic {
      A653_SET_PRIORITY(INVALID_PROCESS_ID, HIGH_PROCESS_PRIORITY, &(PMaster_Test->ret));
    }


// Expected RETURN_CODE == INVALID_PARAM
    atomic {
        c_code{
            if ( PMaster_Test->ret == INVALID_PARAM)
            {
                printf("T-API-PROC-0340:0010  =PASS\n");
            } else
            {
                printf("T-API-PROC-0340:0010  =FAIL\n");
            }
assert(go);
        }
    }


    atomic {
        A653_STOP_SELF(PMaster_Test->_pid);
    }
}


//////////////////////////////////////////////
//////Init:
//////////////////////////////////////////////


c_code {
    PROCESS_ATTRIBUTE_TYPE  a_att;
    PROCESS_ID_TYPE  a_proc;
    RETURN_CODE_TYPE  a_return;
};
c_track "&a_att" "sizeof(PROCESS_ATTRIBUTE_TYPE)" "Matched"
c_track "&a_proc" "sizeof(PROCESS_ID_TYPE)" "Matched"
c_track "&a_return" "sizeof(RETURN_CODE_TYPE)" "Matched"
```

```
init{ atomic {


// Init Model
    c_code {
        MODEL_A653_Init();
    };


// Create & Start Master_Test
    c_code{
        strcpy (a_att.NAME, "Master_Test");
        a_att.STACK_SIZE = 200;
        a_att.BASE_PRIORITY = REGULAR_MASTER_PROCESS_PRIORITY;
        a_att.PERIOD = INFINITE_TIME_VALUE;
        a_att.TIME_CAPACITY= INFINITE_TIME_VALUE;
        a_att.DEADLINE = HARD;
    };


    A653_CREATE_PROCESS(&a_att, &a_proc, &a_return,eso2);
    A653_START(a_proc,&a_return);


// Create & Start P1
    c_code{
        strcpy (a_att.NAME, "P1");
        a_att.STACK_SIZE = 200;
        a_att.BASE_PRIORITY = HIGH_PROCESS_PRIORITY;
        a_att.PERIOD = INFINITE_TIME_VALUE;
        a_att.TIME_CAPACITY= INFINITE_TIME_VALUE;
        a_att.DEADLINE = HARD;
    };


    A653_CREATE_PROCESS(&a_att, &a_proc, &a_return,eso2);
    A653_START(a_proc,&a_return);


// Start Scheduling
    A653_SET_PARTITION_MODE(NORMAL,&a_return);
    run clock();
}; };
```