Logic Model Checking of Time-Periodic Real-Time Systems

Mihai Florian, Ed Gamble, Gerard Holzmann
NASA/JPL Laboratory for Reliable Software,
California Institute of Technology, Pasadena, CA, 91109, USA

In this paper we report on the work we performed to extend the logic model checker SPIN with builtin support for the verification of periodic, real-time embedded software systems, as commonly used in aircraft, automobiles, and spacecraft. We first extended the SPIN verification algorithms to model priority based scheduling policies. Next, we added a library to support the modeling of periodic tasks. This library was used in a recent application of the SPIN model checker to verify the engine control software of an automobile, to study the feasibility of software triggers for unintended acceleration events.

I. Introduction

Real-time embedded systems with well-defined periodic tasks are used in many aerospace applications, as well as in safety critical applications for automotive and medical devices. The correctness of such systems is of significant importance because malfunctions can induce notable economic costs, and they can even cause the loss of human life. Testing is not sufficient to establish the correctness of these systems because it can cover only a minute fragment of all the possible behaviors. An error can be present in any untested behavior and might manifest itself only after the system has been deployed. To verify the correctness of such systems more rigorously than is possible with conventional testing, it can be attractive to apply powerful logic model checking tools that were designed to handle what are fundamentally non-deterministic, multi-tasking systems.

An explicit state model checker such as SPIN[1] uses a finite state model of a system and a formal correctness requirement about feasible and infeasible computations, and it then automatically verifies if the model satisfies the requirement. The model of the system is often given in the form of a finite state automaton written in the specification language of the model checker. The correctness claim is expressed in logic, usually linear temporal logic (LTL).[4]

There is a broad range of model checking tools available, but to date none of these tools includes direct support for modeling and verifying real-time embedded systems with periodic tasks.

Direct support for this domain of application can be important for at least two reasons. First, by having domain specific primitives predefined in the specification language, models can be developed faster, and the user can be more confident to capture the correct semantics of the embedded operating system. Second, builtin support can provide an effective way to deal with the potential complexity of the model checking problem: the number of states of a systems can grow very rapidly with problem size. Having domain specific knowledge, the model checker is able to use optimizations that can limit the complexity increase.

SPIN is one of the leading logic model checking tools available to date. It has been distributed freely since approximately 1990, and enjoys thousands of active users worldwide. Our work has therefore targeted the extension of this system.

We extended the SPIN verification algorithms to accurately model priority based schedulers, as commonly used in real-time embedded systems. This extension is included in the SPIN distribution starting with version 6.2.0[a]. This extension provides language primitives for defining and updating process priorities. The SPIN model checker exploits the extra information about the process priorities and avoids exploring executions that, according to the scheduling policy, are not feasible and thus reduces the computational complexity of the verification.

[a]available from http://spinroot.com/

We also developed a support library to facilitate the modeling of periodic tasks. This library provides key abstractions for modeling alarms and tasks as used in real-time embedded operating systems. The system's periodicity is modeled as a repeating *major cycle* which in turn can define an arbitrary, fixed, number of *minor cycles*. The support library is designed to minimize state variations that could otherwise negatively impact the computational complexity of verification attempts. Correctness claim patterns are defined whereby the computations associated with alarms and tasks either must or need not be completed by the end of a major cycle. The initial motivation for the library was to emulate OSEK[6],[2] a common real-time operating system used for embedded systems development (cf. http://en.wikipedia.org/wiki/OSEK).

We provide examples of how to use these two extensions. First we show how the priority inversion problem can be modeled and solved using the first extension. Then, we describe how we used the second extension to study the feasibility of software triggers for unintended acceleration events in the engine control software of a car.

II. Spin Extension for Modeling Priority-based Schedulers

II.A. Using SPIN

SPIN is an efficient explicit state model checker that can be used to verify models written in the ProMeLa (an acronym for: Process Meta Language). Although this specification language itself supports a range of high-level concepts, such as non-determinism and communication via abstract channels, it can be extended effectively with fragments of implementations that are written directly in the C programming language. Especially this latter feature can significantly simplify the effort required in modeling complex software systems. SPIN models consist of a set of concurrent processes that can communicate and synchronize using shared variables and channel-based message passing. In most model checking systems that have been developed, the number of processes that make up a system is defined statically and cannot change. SPIN was the first system that allows processes to be created (or to disappear) also dynamically, which is a critical feature for modeling real-time embedded systems.

The basic modeling language secures that each system process that is modeled is equivalent to a finite state automaton (although potentially a very large one). Processes are composed asynchronously, based on standard interleaving semantics, to form the system automaton. Because the total number of processes is bounded, the system model as a whole also remains finite state, which guarantees that all properties of interest are in principle decidable for SPIN models. The system automaton is constructed by the tool and simultaneously verified with an on-the-fly verification procedure. A range of different exploration algorithms are supported, including standard depth-first and breadth-first search, with both sequentially executing and parallel executing variants (which can provide a very notable speedup of the verification procedure on generally available multi-core hardware). Visited states are stored in a state space set, to prevent the same basic system states from being explored repeatedly. SPIN furthermore supports a variety of lossless and lossy state compression methods that can enable efficient searches of what otherwise would be intractably large problem sizes.

Besides assertion failures, unreachable code, and deadlocks (jointly known as *safety properties*), SPIN also supports the verification of a much broader class of ω-regular properties (which include so-called *liveness properties* and all properties that can be defined in linear time temporal logic, or LTL). To verify an LTL property ϕ, SPIN constructs the Buchi automaton that corresponds to $\neg\phi$. The language accepted by this automaton is the set of all behaviors that violate the property ϕ. SPIN then constructs on-the-fly the automaton that accepts the intersection of the set of all possible system behaviors (accepted by the system automaton) with the set of all behaviors that violate the property ϕ. If the language accepted by the resulting automaton is empty then the system satisfies the property ϕ, otherwise any word that was accepted by the automaton is a counterexample.

II.B. Modeling Process Priorities

We extended the ProMeLa syntax to allow the specification of process priorities used in real-time embedded software. Each process now defines a local variable `_priority` which records the current priority of the process. Priorities are integer numbers in the interval [1, 255], with 1 being the lowest possible priority and 255 the highest.

Initial priorities are specified using the `priority` keyword (followed by an integer value in the range given

above) immediately after the argument list of a proctype declaration or immediately after the arguments of a run expression can be used that create new processes dynamically.

The declaration

```
active proctype P() priority 10
{  ...
      run P() priority 34;
   ...
}
```

for instance, creates a process named `P` that is active in the initial system state and that executes with an initial priority of 10. Process `P` at some point will execute the statement

```
run P() priority 34
```

which creates a new instance of the same process type, but this time the newly created instance will execute with initial priority 34. If the priority keyword is omitted then the default initial priority of a process is 1.

A process can change its own priority by assigning a new value to the predefined local variable `_priority`.

It can also be useful to inspect or modify the priorities of other processes. To *get* the priority of a process with identifier `p` (the process *pid*), SPIN version 6.2 provides a new predefined function `get_priority(p)`. Similarly, to *set* the priority of a process with pid `p` to the value `v`, SPIN provides a new predefined function `set_priority(p,v)`. The arguments `p` and `v` can be any valid ProMeLa expression, provided that they yield values in the proper range: the process with pid `p` must exist, and the new priority value must be in the valid range [1..255].

The new support for process priorities differs importantly from earlier support that applied only to process simulations. In earlier versions of SPIN, process priorities only determined the likelihood that a process would be scheduled for execution during random simulation runs. For backward compatibility, this earlier behavior can be restored with SPIN option flag `-o6`. The new semantics for priorities now apply equally to simulation and verification, and match the actual semantics of process scheduling priorities as defined in systems like OSEK and VxWorks, etc.

The SPIN verification algorithms, finally, were extended to support the new semantics of the priority keywords and the added functions that can manipulate process priorities dynamically. If process priorities are used, then only those enabled processes that have the highest executing priority among the enabled processes can effectively execute. This can reduce the number of interleavings that the model checker has to explore when looking for requirement violations, and can thus help in an important way in reducing verification complexity for embedded systems software. The extensions are not compatible, though, with other builtin optimization algorithms that SPIN normally uses to reduce the cost of verification. One such algorithm is its partial order reduction strategy. When priorities are used, the partial order reduction rules are no longer valid and the optimization must be disabled. One further restriction is that the new process priority semantics is not compatible with rendezvous operations between processes of different priorities (i.e., synchronous communication). The latter restriction may be removed at some point, once the underlying theory for new reduction rules has been worked out.

III. The Task Library

A small ProMeLa library file supports the modeling[b] and the verification of real-time embedded systems with periodic behavior. The library is available as a ProMeLa header file (task.pml) that when included in other ProMeLa models provides the abstractions of a small real-time operating system (e.g. OSEK/VDX[6]), most notably task and alarm management routines. We designed the task library to minimize the impact on verification complexity, for instance, by avoiding the storage of non-essential local state.

The task library models the periodicity of a system based on a repeating *major cycle*. Each major cycle consists of a fixed number of *minor cycles*, also called *ticks*. The number of ticks per major cycle is set by defining the `TICKS_IN_CYCLE` macro before including the task library in the model. We have not attached any time units to the ticks and each model that uses the task library can interpret the duration of a minor cycle as appropriate. This allows the users of the task library to control the time granularity of the model.

[b] available on request from the authors

For example, a system operating at 100 Hz might be modeled as having a major cycle every 10 ms with 1 ms ticks, but it can also be modeled as having a major cycle every 10 ms with 5 ms ticks (for 2 ticks per major cycle).

To minimize the impact on the size of the search space that the model checker explores during verifications, the internal state of the task library resets on every major cycle. The number of ticks in a major cycle can have a large impact on the number of states of the system. Thus a model with 10 ticks per major cycle can have up to 10 times the computational complexity of a model with 1 tick per major cycle. Of course, specifics of the system's behavior, both at the tick level and the cycle level are also contributing factors to the verification's complexity, but because it acts as a multiplier, the number of ticks in a cycle should be kept at the smallest value that can still ensure accuracy.

The two key abstractions provided by the task library are *tasks* and *alarms*. The number of tasks and alarms can be set by providing the appropriate values for the macros `NUMBER_OF_TASKS` and `NUMBER_OF_ALARMS` before including the task library. To keep the size of a state manageable, the current version of the task library allows only up to 5 simultaneous tasks and alarms. SPIN itself can support the modeling of up to 255 distinct asynchronous processes, with up to 255 independent channels of communication between them.

III.A. Tasks

A task is the primary unit of computation in a periodic, real-time embedded system, and thus it is natural for tasks to be mapped to ProMeLa processes. The mapping is achieved by calling the `task_setup` primitive from within a Promela proctype. Tasks can be either *periodic* or *event-based*.

An event-based task `tid` is triggered by calling the `ActivateTask(tid)` primitive. If the task was already active then the call has no effect. Otherwise, the task becomes active starting with the next tick. A task `tid` that is not active can wait to be triggered by calling the `WaitTask(tid)` primitive. A task `tid` that has terminated its computation must call the `TerminateTask(tid)` primitive before it can be activated again.

A periodic task with period p triggers once every p major cycles at a specified phase offset (measured in ticks). Just like an event-based task, a periodic task `tid` has to wait for its activation by calling the `WaitTask(tid)` primitive. Also, when a periodic task completes its computation it must call the `TerminateTask(tid)` primitive so it can be triggered again in a subsequent major cycle. From a modeling perspective the phase offset provides a means to sequence tasks. There is, however, no guarantee where precisely any given task runs relative to any other task of the same priority level. Equal priority tasks that must coordinate their executions can do so with the help of standard ProMeLa primitives, outside the task library.

A task can be *required* or *not required*. A required task must complete before the next major cycle can start. The task library delays the beginning of the next cycle by extending the last tick of the current cycle until all the required tasks terminate. This delay in the tick can be of no temporal consequence because ProMeLa and the task library have no actual notion of time. From a system modeling perspective, a task that is required is deemed to have sufficient margin per major cycle whereas a task that is not required might have a questionable level of margin.

Below is an example of how a required periodic task can be modeled as a standard ProMeLa process, with the use of the new task library:

```
1   active proctype task_proc () {
2     byte _tid;
3     /* ...local state can be defined and initialized here... */
4     task_setup(_tid, true, 1, 0, true, true);
5     system_init_is_complete ->
6   end:
7     do
8     :: WaitTask (_tid) ->
9        /* ...computation... */
10       TerminateTask(_tid)
11    od
12  }
```

The `task_setup` in line 4 registers the current process as a task with the task library. The prototype of the call is `task_setup(tid, is_periodic, period, phase, is_required, initialized)` and thus line

4 registers the current process as a task that is periodic, with period 1, at phase offset 0, that is required to complete by the end of the major cycle. If the `initialized` flag is set then any initialization for the taks must have happened before the call to `task_setup`. If the `initialized` flag is `false` then some extra initialization can be done after the call to `task_setup`. In this case the task library must be informed when the initialization is complete by executing `TASK_IS_INITIALIZED(_tid) = true`.

Other types of tasks (event based or not required) can be set up by passing the appropriate arguments to `task_setup`.

After a task is set up, it has to wait for the whole system to be initialized before it can do anything (this happens on line 5).

The `do` loop in line 7 is required so that the task can be activated on every major cycle. This loop is also required for an event based task if we want to activate the task multiple times. The `end` label informs SPIN that the process is allowed stay in the `do` loop forever.

After all the tasks have been defined, the system as a whole can be initialized as follows:

```
1  init {
2    byte index;
3    /* the global environment can be set up here */
4    tasks_init ();
5    system_init_can_start = true;
6    WAIT_ON_TASK_INITIALIZATION && WAIT_ON_ALARM_INITIALIZATION ->
7    system_init_is_complete = true;
8  }
```

On line 6 the init process waits for all the tasks and alarms to be initialized before it signals that the system can start (line 7).

Periodic and event-based tasks with their configuration options provide the flexibility to model a wide variety of real-time processing behaviors.

III.B. Alarms

Alarms are an important concept in real-time embedded systems. The alarms in the task library provide a mechanism to wait for a tick and then to perform actions specific to that tick. An alarm `aid` is initialized by calling the `alarm_setup(aid)` primitive. A subsequent call to the `AlarmWait(aid,ticks)` primitive blocks the Promela proctype until the alarm triggers at the next tick. Once triggered, the `ticks` variable is set to the value of the current tick in the current major cycle.

Just like a task, an alarm can be modeled as a ProMeLa process. Usually the alarm processes are distinct from the task processes. Below is an example of how the mapping can be done:

```
1  active proctype alarm_proc ()
2  {
3    aid  _aid;
4    tick _ticks;
5    alarm_setup (_aid);
6    system_init_is_complete ->
7  end:
8    do
9    :: AlarmWait(_aid, _ticks) ->
10      /* computation for the current tick */
11   od
12 }
```

Because alarms are triggered on every tick, they can be used to model the environment of the embedded system. A common scenario of using alarms is to monitor the state of the system in each minor cycle and activate event-based tasks when certain conditions are satisfied, thus simulating the generation of interrupts.

IV. Application

IV.A. Modeling Priority Inversion

The SPIN extensions for supporting priority based scheduling can be used to model the *priority inversion* problem[5] that occurs in systems that use priority based schedulers when processes of different priorities compete for shared resources. The problem achieved instant fame when the software in the Sojourner rover reached a deadlock on the surface of Mars in 1997 when a priority inheritance feature had accidentally been disabled.

We can illustrate the problem with a system of just three processes: a low priority process, a high priority process, and a medium priority process. Consider the scenario where the low priority process executes and grabs a shared resource, e.g., by obtaining a semaphore or a mutual exclusion lock. Before the low priority process can complete its execution and release the resource, the high priority process becomes active and request the same resource. Because the resource is not available, the high priority process is blocked and the low priority process continues to execute. But in the scenario of interest, before the low priority process can release the resource, a medium priority process (not requiring access to the shared resource) becomes active and preempts the low priority process, which is now suspended and is no longer able to release the resource. Because the low priority process is preempted and the high priority process is blocked waiting for the shared resource, the medium priority process can execute indefinitely. A priority inversion occurred between the medium priority process and the high priority process even though they do not compete for the shared resource.

This scenario is exemplified in the following SPIN model. To illustrate the problem, we use a mutual exclusion lock `mutex` as the shared resource.

```
1   mtype = { free, busy };
2   mtype mutex = free;
3
4   inline lock()   { atomic { mutex == free -> mutex = busy } }
5   inline unlock() { mutex = free }
6
7   proctype high() {
8       lock();
9       /* critical section */
10      /* unreachable, because medium starves high */
11      assert(false);
12      unlock()
13  }
14
15  proctype medium() {
16  end: do :: true -> skip od
17  }
18
19  active proctype low() priority 3 {
20      lock();
21      run high() priority 7;
22      run medium() priority 5;
23      /* unreachable because medium preempts low */
24      /* critical section */
25      unlock();
26      assert(false)
27  }
```

To reproduce the problem in this small example, the low priority process starts the high and medium priority processes (lines 21 and 22) after it has grabbed the mutex (line 20).

When the model is verified with SPIN it proves that the assertion violations on lines 11 and 26 are unreachable. The high priority process never succeeds in obtaining the mutual exclusion lock (reaching line 9) and the low priority process is never able to release it.

To fix the above problem a *priority inheritance*[5] mechanism can be adopted: with this feature the process owning the resource has its priority raised to the highest priority of a process waiting for the resource. When the resource is released the priority of the process reverts to its original value. With the extended version of SPIN we can directly model this feature. The required changes to the `lock()` and `unlock()` primitives are illustrated in the following model. The remainder of the model stays the same.

```
1   mtype = { free, busy };
2   mtype mutex = free;
3   byte mutex_owner; /* pid of the process that locked the mutex */
4   byte mutex_priority; /* the original priority of the process that locked the mutex */
5
6   inline lock() {
7       atomic {
8           do
9           :: mutex == free ->
10             mutex = busy;
11             mutex_owner = _pid;
12             mutex_priority = _priority;
13             break;
14          :: else ->
15             if
16             :: get_priority(mutex_owner) < _priority ->
17                    set_priority(mutex_owner, _priority);
18                    mutex == free
19             :: else
20             fi
21          od
22      }
23  }
24
25  inline unlock() {
26      mutex = free;
27      mutex_owner = 0;
28      _priority = mutex_priority
29  }
```

With these changes SPIN immediately detects the now possible assertion violations, thus proving that the lowest priority process releases the resource and the highest priority process can indeed acquire access to it.

IV.B. Modeling Engine Control Software

We have used the new task library from Section III to model the engine control software of a car, as part of a comprehensive NASA study of the feasibility of software triggers for unintended acceleration events. The study was performed at the request of the US Department of Transportation. The SPIN model we developed to study this problem consisted of one alarm and two tasks, each mapped to a separate ProMeLa process.

The alarm simulated the environment for the two system tasks. One task was event-based and corresponded to an interrupt handler. The other task was periodic and performed a speed computation that provided a critical input to the computation of a throttle opening, which in turn determines engine speed. The major cycle in this model was chosen to be 4 ms long and had 4 ticks (for 1 ms per tick).

The alarm process maintained an internal representation of the speed and acceleration of the car and simulated the actions of the driver. Every 250 ms, for example, the speed was either increased or decreased. Based on this model, the alarm process simulated the generation of pulse interrupts by calling `ActivateTask` to trigger the interrupt handler task. The alarm also updated a timer register that counted how many ticks passed since the last pulse. The frequency of the pulses was used as an indication of the current speed of the car (or its engine).

The interrupt handler task processed pulse events and was activated each time a pulse interrupt was generated by the alarm process. The purpose of this task was to count in a register how many pulses arrived in the current speed computation time window.

The speed computation task ran once every 4 ms. Most of the time it did not do much more than to count how much time has passed from the current speed computation time window. Every 471 ms the speed computation task closed the speed computation time window and used the number of pulses that had arrived in the time window (counted by the interrupt handler task) to update the estimate of the current speed of the car. The number of pulses in the last time window was saved so that it could be used in future speed estimations.

The property we verified in this case stated that if the current speed in the model from the alarm process remained constant then the speed estimated by the speed computation task remained within a confidence interval and could not change suddenly. This corresponded to a scenario where the driver tries to keep the car at a constant speed and it ensures that the car cannot suddenly gain or lose speed without the driver's intervention. Our verification did not establish a possible cause for unintended acceleration that could be represented within the accuracy of our models. The full results of this study were published in a comprehensive report and can be obtained freely, though alas many details of the study had to be redacted.[3]

For the future, we plan to update the task library to model with additional primitives that may prove useful in modeling real-time embedded operating system software.

V. Conclusions

We extended the SPIN model checker to enable direct modeling and verification of periodic, real-time embedded software systems. We added support for priority based schedulers and we developed a task library that provides some of the primitives of a real-time embedded operating system.

We illustrated our extensions in this paper with a few small examples, including the modeling of the priority inversion problem and the use of priority inheritance to prevent it. We also briefly discussed a larger study of the feasibility of software triggers for unintended acceleration events in the engine control software of a car, which was performed at the request of NASA and the US Department of Transportation.

Acknowledgments

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors are grateful to NASA and the NHTSA for presenting us with a challenging and interesting application of static analysis and logic model checking techniques. We also thank JPL's Mark McKelvin for help with the documentation of the new SPIN primitives and task library.

References

[1]Holzmann, G. J., *The SPIN model checker: primer and reference manual*, Addison-Wesley, 2004.

[2]McKelvin, M., and Holzmann, G.J., *Model checking multitask applications for OSEK compliant real-time operating systems*, Proc. 17th IEEE Pacific Rim Int. Symp. on Dependable Computing (PRDC 2011), Pasadena, CA, Dec. 12-14, 2011.

[3]NHTSA, Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation, Appendix A. Software, Jan. 18, 2011, 134 pgs. `http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA_FR_Appendix_A_Software.pdf`

[4]Pnueli, A., *The Temporal Logic of Programs*, 18th Annual Symposium on Foundations of Computer Science (FOCS), Oct. 1977, pp. 46-57.

[5]Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority inheritance protocols: an approach to real-time synchronization*, IEEE Transactions on Computers, Volume 39, Issue 9 (1990), pp. 1175-1185.

[6]Zahir, A., Palmieri, P., *OSEK/VDX-operating systems for automotive applications*, OSEK/VDX Open Systems in Automotive Networks, Nov. 1998.