DEOS Kernel: Environment Modeling using LTL Assumptions

Corina Pasareanu
RIACS
NASA Ames Research Center
Moffet Field, CA 94035

July 5, 2000

Abstract

We report about our attempt to automatically construct an environment for the DEOS kernel (used while model checking the kernel). The kernel was analyzed in [3, 4], and a bug was discovered. The authors of [3, 4] built the environment by hand.

We successfully use the methods described in [1, 2]: we start with the most general definition of the environment, we establish a set of LTL environment assumptions and we use these assumptions to refine the environment definition. Using the refined environment, we discover the error that is reported in [3, 4]. Moreover, the environment is precise enough s.t. when used with the corrected version of the kernel, no "false errors" are reported.

Introduction

The Honeywell Dynamic Enforcement Operating System (DEOS) [3] is a real-time operating system for integrated modular avionics systems. In previos work [3], the Spin model checker was used to verify the time partitioning in the DEOS real-time scheduling kernel and to detect a subtle implementation error that was originally discovered and fixed during the standard formal review process.

The most difficult task in the experiment in [3] was constructing an adequate environment for the kernel to execute in. In the case of DEOS, it was necessary to construct models of the possible behaviors of user threads (that run on the kernel), the system clock and the system timer (that generate hardware interrupts). Modeling the generation of interrupts was the most difficult part of constructing the environment for DEOS. Promela and Spin do not handle real time, so the passing of time had to be modeled explicitly.

Our study is aimed at automatically constructing the environment that models the generation of interrupts from a set of LTL constraints (or filters) as described in [1, 2], as opposed to the ad-hoc method used in [3].

We believe that our study shows that the methods from [1, 2] provide a simple and elegant solution to the problem of environment construction. As stated in [3], the construction of the environment is a serious problem that has to be solved, in order to use model checking to find errors in "real" programs.

We proceed by surveying some background material. Then, a very short description is given of the DEOS kernel and its environment as presented in [3, 4]. We show how we build the environment systematically, starting with the universal environment, and with the environment assumptions. We present the results of using Spin to model check the DEOS kernel with the new environments. And we write our conclusions and indicate some directions for future work.

Background: Assume-Guarantee Model Checking with Spin

To model check a system, an environment must be constructed that drives the program [1, 2]. The most general environment for properties stated in LTL is the "universal" environment, that is capable of executing any sequence of operations in the system's interface. Under the assume-guarantee paradigm, LTL environment assumptions can be exploited to constrain the definition of the environment. In particular, if the assumption ϕ and the guarantee ψ are LTL formulas, one can simply check the formula $\phi \rightarrow \psi$ using SPIN. LTL assumptions can also be used to synthesize refined environments, in which case ϕ is eliminated from the formula to be checked (see e.g. [2]).

In our experiment, we first constructed a "universal" environment and then we defined LTL formulas that restrict the behavior of the universal environment.

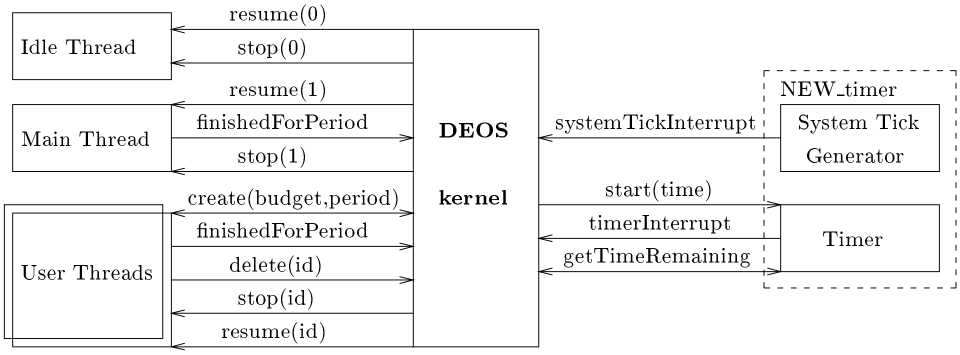

Figure 1: DEOS Kernel and environment

The DEOS Scheduler Kernel and its "Original" Environment

The DEOS kernel together with its environment is illustrated in Figure 1 (simplified from [3]). There is a box for each concurrently executing process: the kernel, the idle thread, the main thread, the user threads to be scheduled, the system tick generator and the timer process. Labeled arrows signify communications between processes. The dotted box indicates that the system tick generator and the timer were combined into one process (*NEW_timer*).

DEOS threads can run in different *scheduling periods* and require a certain amount of CPU time to be allocated to them, called their *budget*, within their period. The scheduling period and budget of each thread is fixed at thread creation. If a thread asks for more budget than is available, the thread will not be created. During execution, if a thread uses more time than its remaining budget then it will be interrupted by the kernel and a new thread will be scheduled.

A detailed description of the DEOS kernel and of its environment can be found in [3, 4]. The description of the DEOS kernel was of no interest to us, since one of the purposes of this study was to construct the environment without looking at the kernel, but just using the kernel's interface. Figure 2 shows the Promela code for *NEW_timer* as it appears in [4].

The timer model keeps track of the time that has been used in a period and makes sure that a system tick interrupt only occurs when the appropriate amount of time has been used (and vice versa). Variable *uSecsInFastestPeriod* stores the amount of time between system ticks; *Start_time* is assigned the value received from the kernel with which the timer is started; *Remaining_time* stores the time remaining from the thread's budget.

The main aspect of DEOS that was analyzed in [3] was the time partitioning property: each thread in the kernel is guaranteed to have access to its complete CPU budget during each scheduling period. According to [3], a necessary (but not sufficient) condition of time

```
proctype NEW_timer()
{  byte Used_time     =0; /* time used in period since last tick; <= uSecsInFastestPeriod */
   byte Start_time    =0; /* time the timer was started with */
   byte Remaining_time=0; /* time remaining after timer counted down from Start_time */
   byte Y=0; /* Timed used by a thread in one run */

   bool tick_since_start = FALSE;
   bool started=FALSE;
   bool timer_went_off = FALSE;
   do
   ::Sched2Timer?start(Start_time) ->
       tick_since_start = FALSE;
       started = TRUE;
       timer_went_off = FALSE;
   ::Sched2Timer?getTimeRemaining,0 ->
        started = FALSE;
        if
        ::tick_since_start ->
            Timer2Sched[1]!timeRemaining,Remaining_time;
        ::timer_went_off ->
            assert(Remaining_time == 0);
            Timer2Sched[1]!timeRemaining,Remaining_time;
        ::else ->
            /* time used by thread Y: 0 <= Y <= uSecsInFastestPeriod - Used_time AND */
            /*                        0 <= Y <= Start_time                          */
            if
            ::(uSecsInFastestPeriod - Used_time) <= Start_time ->
                Y = uSecsInFastestPeriod - Used_time;
            ::Y = 0;
            fi;
            Remaining_time = Start_time - Y;
            Used_time = Used_time + Y;
            Timer2Sched[1]!timeRemaining,Remaining_time;
        fi;
     /* channel array trick to avoid unnecessary deadlocks */
   ::Tick2Sched[((Start_time+Used_time)>=uSecsInFastestPeriod) && started]!tickintrpt,0 ->
        Y = uSecsInFastestPeriod - Used_time;
        Remaining_time = Start_time - Y;
        Used_time = 0;
        tick_since_start = TRUE;
     /* channel array trick to avoid unnecessary deadlocks */
   ::Timer2Sched[started]!timerintrpt,0 ->
        Remaining_time = 0;
        Y = Start_time;
        Used_time = (Used_time + Start_time)%(uSecsInFastestPeriod +1);
        timer_went_off = TRUE;
   od
}
```

Figure 2: Timer model from previous DEOS verification study

partitioning is: if there is slack in the system (i.e. the main thread does not have 100% CPU utilization), then the idle thread should run during every longest period.

The property (ψ) is written in LTL as:

```
[](beginperiod -> !endperiod U (idle || []!endperiod)),
```

which says that it is always the case that, when the longest period begins, it will not end until the idle thread runs (or it will never end).

Verification was run for a system configuration with two user threads. Spin reported an error scenario where user thread 1 deletes itself (before being interrupted) at the end of the first short period of time. At this point, its budget is given to the main thread. The scheduling then continues normally to the end of the long period boundary. At this point, Spin signals an error because the idle thread did not run between the two period one boundaries. The error stems from the fact that when user thread 1 deleted itself, it immediately returned its budget to the main thread. The result is that user thread 2 does not get all of the CPU time it requested. This bug was the same bug that Honeywell had discovered during code inspections.

Universal Environment for DEOS Scheduler

The problem with the code presented in the previous section is that it is hard to read and understand. Another problem is that it is no obvious way to validate it against a real environment.

In order to build the timer systematically, we identified the interface between *NEW_timer* and the scheduler, and we built the following skeleton:

```
proctype skeleton1_NEW_timer()
{ byte Start_time;
  byte Remaining_time;

  do
  :: Tick2Sched[1]!tickintrpt,0; /* system tick generator */

  :: Sched2Timer?start(Start_time);              /*       */
  :: Sched2Timer?getTimeRemaining,0;             /* timer */
  :: Timer2Sched[1]!timeRemaining,Remaining_time;/*       */
  :: Timer2Sched[1]!timerintrpt,0;               /*       */

  :: skip; /* some internal, non-observable action ... */
  od
}
```

The skeleton is capable of invoking any sequence of interface operations. This environment would be good enough to check properties that are not related to time (provided that we use **Point** [1] abstraction for *Start_time* and *Remaining_time*). Since we are interested in a property that is time-dependent, we have to relate the two variables *Start_time* and *Remaining_time*. We do this by introducing a third variable, *clock*, that records the time remaining in a period. When a tick interrupt occurs, *clock* is reset, as it would happen in a real system.

When the timer is started, with value specified in *Start_time*, the remaining time is estimated:

- If *Start_time* is greater than the current value of *clock*, and a system tick interrupt occurs in the future, then *Remaining_time* for the current thread will be greater than zero, and the *clock* will be set to zero, which signifies the end of the current short period.

- If *Start_time* is less than or equal to the current value of *clock*, and a timer interrupt occurs in the future, the *Remaining_time* for the current thread will be zero and the *clock* will be decreased.

- Nondeterministically, the environment can update *Remaining_time* to be *Start_time* and leave *clock* unchanged, which corresponds to the situation that the thread consumed no time.

The resulting code for the timer is shown in Figure 3. Note that the *half-time option* [3, 4], could be easily incorporated in the *if* statement.

Environment Assumptions

When we tried to check the timing property using *skeleton2_NEW_timer*, Spin reported counterexamples that were not valid executions of the system. As noted in [1, 2], universal environments can yield "false" counterexamples, since they over-approximate the behavior of real environments. In our study, it was quite easy to interpret the counterexamples, since we had only to analyze the message exchanges between *NEW_Timer* and *Kernel*, which are nicely displayed by XSpin.

Specifically, in the counterexample, the timer allows period ticks (and resets of the clock) to occur arbitrarily. The problem is that the *clock* should not be reset unless its value is zero (as it would happen in a real system clock). This assumption could be encoded in LTL as:

```
[](tickinterrupt -> !new_tickinterrupt U (clock==0 || []!new_tickinterrupt))
```

which says that after a system tick interrupt occurs, a new tick interrupt can not occur unless the value of the *clock* is zero. We noticed that this assumption can be easily encoded in *NEW_timer*, by restricting the rendez-vous on *tickintrpt* to occur only when *clock* is zero, as shown in Figure 4.

```
proctype skeleton2_NEW_timer()
{ byte clock=0;
  byte Start_time;
  byte Remaining_time=0;

  do
  :: Tick2Sched[1]!tickintrpt,0 -> clock=uSecsInFastestPeriod; /* reset */

  :: Sched2Timer?start(Start_time);
        /* estimate Remaining_time */
        if
        :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
        :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
        :: Remaining_time = Start_time;
        /* put half_time_option here */
        fi;
  :: Sched2Timer?getTimeRemaining,0;
  :: Timer2Sched[1]!timeRemaining,Remaining_time;
  :: Timer2Sched[1]!timerintrpt,0;
  od
}
```

Figure 3: Timer skeleton with *Remainaing_time* variable

```
proctype U_NEW_timer()
{ byte clock=0;
  byte Start_time;
  byte Remaining_time=0;

  do
  :: Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod; /* reset */

  :: Sched2Timer?start(Start_time) -> start:skip;
        /* estimate Remaining_time */
        if
        :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
        :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
        :: Remaining_time = Start_time;
        /* put half_time_option here */
        fi;
  :: Sched2Timer?getTimeRemaining,0;
  :: Timer2Sched[Remaining_time>0]!timeRemaining,Remaining_time ->timeRemainingGT0:skip;
  :: Timer2Sched[Remaining_time==0]!timeRemaining,0;
  :: Timer2Sched[1]!timerintrpt,0 -> timerinterrupt:skip;
  od
}
```

Figure 4: Timer with restricted rendezvous on *tickintrpt*

We attempted to check the timing property, using the restricted environment and again Spin gave a counter example that was not a valid execution of the system. In the error trace, after a timer interrupt occurs and the kernel asks for the remaining time, the value returned is greater than zero (which shouldn't happen in a "realistic" environment, since a timer interrupt signifies that there is no remaining time left for the current thread).

This observation leads to the following assumption (ϕ):

```
[](timerinterrupt -> !timeRemainingGT0 U (start || []!timeRemainingGT0))
```

which says that after a timer interrupt occurs, and the kernel asks for the remaining time, the environment can not return a value greater than zero, unless the timer is started again (notice in the code below that we inserted some labels to define the predicates and that we split the rendez-vous based on the returned value of *Remaining_time*, e.g. predicate *timeRemainingGT0* is true when the timer thread is at label *timeRemainingGT0*).

When using *U_NEW_timer* and assumption ϕ as a filter, i.e. when checking the combined formula $\phi \rightarrow \psi$:

```
([](timerintrpt -> !timeRemainingGT0 U (start || []!timeRemainingGT0))) ->
      [](endpulse -> !beginpulse U (idle || []!beginpulse))
```

we obtained a "real" counterexample, similar to the one described in [3, 4] (the assumption "filtered out" traces that do not correspond to real executions of the system).

Results

We model checked the timing property on the kernel "closed" with *U_NEW_timer* (and the LTL assumption encoded in the formula), and, alternatively, with the environment (automatically) synthesized from the LTL assumption ϕ, as described in [2]. The graph synthesized from the assumption is shown in Figure 5.

The labels on the edges denote the allowed interface operations (i.e. rendez-vous between environment and kernel, together with the code to be executed for each rendez-vous). E.g. label *systemTickInterrupt* stands for:

```
Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod;
```

Node 0 designates the initial state. The corresponding Promela code (*S_NEW_timer*) is given in the appendix.

We used Spin version 3.2.5a on a SUN ULTRA60 with 1G of RAM. The following table gives data for each of the model-checking runs (using *U_NEW_timer*, *S_NEW_timer* and the original *NEW_timer*). We report the total of user and system time in seconds to convert LTL formulas to the Spin input format, i.e. never claim, (t_{never}) and to execute the model checker (t_{MC}), we also report the memory used in verification in Mbytes (*mem*) and the size of the shortest error trace.

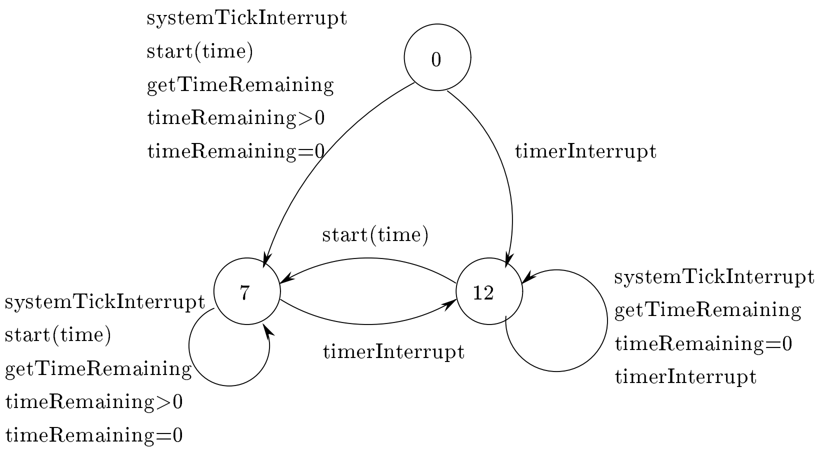

Figure 5: Synthesized assumption graph

Environment	t_{never}	t_{MC}	*mem*	*error trace depth*
U_NEW_timer	1:49.97	1.3	3.633	1988
S_NEW_timer	0.1	0.1	2.609	1554
NEW_timer	0.1	0.1	2.609	1619

In conformance with the data from [2], synthesized environments enable faster model checking and better use of memory. The time for generating the never claim with the assumption encoded into the formula to be checked is the dominant time.

We model checked the corrected kernel, and Spin exhaustively searched the state space, and reported the following data; no (false) errors were found.

Environment	t_{MC}	*mem*
U_NEW_timer	1:38.1	102.289
S_NEW_timer	8.8	23.172
NEW_timer	2.9	12.996

The original environment and the synthesized environment enable similar uses of time and memory. We note that we could validate the environment built by hand by checking the assumptions against the kernel "closed" with *NEW_timer*.

Conclusions

We have presented an application of the methods from [1, 2], to a "real" case-study: the verification of the DEOS scheduler using Spin. We started with the "universal" environment and we refined it using LTL assumptions, based on "false" counter-examples reported by Spin. Using the refined environment, Spin discovered the same error that was reported in [3, 4]. Moreover, Spin found no errors in the corrected kernel.

It was a nice and easy job that took an effort of several days. We built the environment without looking at the code for the kernel; we examined only the code for the environment, described in [4]. We had the great advantage of looking at code that was previously analyzed. The error was known, and also the configuration of the system (i.e. at least two user threads are necessary in order to find the error). We derived the environment assumptions from the "false" counter-examples displayed by XSpin. We note that for the environment built in [3, 4], there was no clear specification of the assumptions (they were implicitly hard-coded).

We note that our approach is not entirely automatic. We introduced variable *clock* to relate variables and we also wrote one of the assumptions directly into the code, because it was more natural than expressing it as an LTL filter.

We didn't use data abstraction [1] for time in the system. It would be interesting to investigate in future work what kind of abstractions could be used and for what kind of properties.

One of the questions for future work, partially answered here, is how much of the environment generation can be done automatically. We believe that our method greatly reduces the amount of user intervention.

Bibliography

[1] M.B.Dwyer, C.S.Pasareanu. *Filter-based Model Checking of Partial Systems.* - FSE'98.

[2] C.S.Pasareanu, M.B.Dwyer, M.Huth. *Assume-Guarantee Model Checking of Software: A Comparative Case Study* - Spin'99.

[3] J.Penix, W.Visser, E.Engstrom, A.Larson, N.Weininger. *Verification of Time Partitioning in the DEOS Scheduler Kernel.* - ICSE'2000.

[4] J.Penix, W.Visser, E.Engstrom, A.Larson, N.Weininger. *Translation and Verification of the DEOS Scheduling Kernel.*

Appendix: Synthesized Code for *NEW_timer*

```
proctype S_NEW_timer()
{ byte clock=0;
  byte Start_time;
  byte Remaining_time=0;

  state_0: if
           :: Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod;goto state_7;
           :: Sched2Timer?start(Start_time) ->
                if
                :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
                :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
                :: Remaining_time = Start_time;
                fi; goto state_7;
           :: Sched2Timer?getTimeRemaining,0 -> goto state_7;
           :: Timer2Sched[Remaining_time>0]!timeRemaining,Remaining_time -> goto state_7;
           :: Timer2Sched[Remaining_time==0]!timeRemaining,0 -> goto state_7;
           :: Timer2Sched[1]!timerintrpt,0 -> goto state_12;
           fi;
  state_7: if
           :: Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod;goto state_7;
           :: Sched2Timer?start(Start_time) ->
                if
                :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
                :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
                :: Remaining_time = Start_time;
                fi; goto state_7;
           :: Sched2Timer?getTimeRemaining,0 -> goto state_7;
           :: Timer2Sched[Remaining_time>0]!timeRemaining,Remaining_time -> goto state_7;
           :: Timer2Sched[Remaining_time==0]!timeRemaining,0 -> goto state_7;
           :: Timer2Sched[1]!timerintrpt,0 -> goto state_12;
           fi;
  state_12:if
           :: Tick2Sched[clock==0]!tickintrpt,0 -> clock=uSecsInFastestPeriod;goto state_12;
           :: Sched2Timer?start(Start_time) ->
                if
                :: Start_time > clock -> Remaining_time=Start_time-clock;clock=0;
                :: Start_time <= clock -> Remaining_time=0;clock=clock-Start_time;
                :: Remaining_time = Start_time;
                fi; goto state_7;
           :: Sched2Timer?getTimeRemaining,0 -> goto state_12;
           :: Timer2Sched[Remaining_time==0]!timeRemaining,0 -> goto state_12;
           :: Timer2Sched[1]!timerintrpt,0 -> goto state_12;
           fi;
}
```