DOI: 10.15514/ISPRAS-2020-32(6)-4

A Formal Model of a Partitioned Real-Time Operating System in Promela

S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
Polzunov Altai State Technical University,
Lenin avenue 46, Barnaul, 656038, Russia

Abstract. Real-time partitioned operating systems meet the current avionics standard of reliable software; they are capable of responding to events from devices with an expected speed, as well as sharing processor time and memory between isolated partitions. Model-based Checking is a formal verification technique in which a software model is developed and then it is automatically checked for the compliance with formal requirements. This method allows proving the correct operation of the model on all possible input data, all possible ways of processes switching and interactions. In this article, we describe a formalized model of an open-source partitioned operating system POK. We implement the model in Promela language for SPIN tool with the purposes of formal verification using the Model Checking method. The model is designed to describe the behavior of: partition and process schedulers, system calls through a software interrupt, kernel libraries for working with synchronization primitives and processes awaiting, user code which consists of several processes in different partitions that are synchronized through a semaphore. The described approach can be used to verify the correct synchronization, the proper operation of the scheduler algorithms, and the accurate data access from different partitions by introducing the corresponding requirements in the form of formulas of the linear-time temporal logic.

Keywords: formal verification; operating system; partitioned system; real-time system; model checking; system programming; Promela; SPIN

For citation: Staroletov S., A Formal Model of a Partitioned Real-Time Operating System in Promela. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 49-66. DOI: 10.15514/ISPRAS-2020-32(6)-4

Формальная модель партицированной операционной системы реального времени на Promela

С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
Алтайский государственный технический университет им. И.И. Ползунова,
656038 Барнаул, проспект Ленина, 46

Аннотация. Текущий стандарт надежного программного обеспечения для бортовых контроллеров – это многоразделъная операционная система реального времени, которая способна реагировать на события от устройств с ожидаемой скоростью, а также делить процессорное время и память между изолированными разделами. Верификация на основе модели – это метод формальной проверки программного обеспечения, при котором разрабатывается программная модель, а затем она автоматически проверяется на соответствие формальным требованиям. Этот метод позволяет доказать правильность работы модели на всех возможных входных данных, всех возможных способов переключения процессов и взаимодействий. В этой статье описывается формализованная модель открытой многоразделъной операционной системы POK, реализованная на языке Promela средства SPIN для формальной верификации методом Model Checking и предназначенная для моделирования поведения: планировщика разделов и процессов; системных вызовов через программное прерывание; библиотеки ядра для работы с примитивами синхронизации и ожиданием процессов; пользовательский код, осуществляющий работу нескольких процессов в разных разделах, которые синхронизируются

через семафоры. Данный подход может быть использован для проверки корректности синхронизации, работы алгоритмов планировщика, корректного доступа к данным из разных разделов путем задания соответствующих требований в виде формул темпоральной логики линейного времени.

Ключевые слова: формальная верификация; операционные системы; партицирование; ОС реального времени; Model Checking; системное программирование; Promela; SPIN

Для цитирования: Старолетов С.М. Формальная модель партицированной операционной системы реального времени на Promela. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 49-66 (на английском языке). DOI: 10.15514/ISPRAS–2020–32(6)–4

1. Introduction

This work is a part of a project to provide verification methods for controllers of cyber-physical systems with high-reliability requirements [1]. In this paper, we refer to a concept of partitioned operating systems, mostly related to avionics software standards. The main goal is to develop and verify software based on existing open-source solutions, as well as to apply the results as a model for teaching the courses *«Components of operating systems»* and *«Software verification»*.

In this paper, we follow the creation of a model for *POK* (Partitioned Operating System Kernel) [2]. Using its source code, we create a corresponding code in *Promela* [3], an input language of *SPIN* verifier. On the one side, the language offers to encode real algorithms close to original C implementation, but on the other side, this language has a clear formal semantic and the model in this language can (without any shortcomings) be translated to a Kripke structure and then verified by querying LTL formulas with temporal properties of desired OS model behavior.

This publication has the following structure: in Section 2, we briefly describe the POK concept and model checking with SPIN; in Section 3, we show the core of presented approach, how to model a client program using an emulation of the instruction pointer; in Section 4, we highlight our scheduling model; in Section 5, we present ways to model the syscalls; in Section 6, we browse some existing solutions in this area; in Section 7, we discuss the solution and finally, in Section 8, we make a conclusion and give a link to our resulting open-source model in Promela.

The main contributions of the paper are: (a) we show the applicability of Promela to model OS behavior; (b) we create an executable model of a partitioned OS.

2. Background

2.1. A Concept of Partitioned Real-time OS

A BSD-licensed open-source OS POK, which satisfies avionics software standards with some limitations, was created at a research institute in France as a PhD thesis by Julien Delange [4], it applies the Model-Driven Engineering approach [5] for describing the system configurations, and its source code is available in [2].

We have already summarized in [1] its main features as:

- *MDE approach*: initial OS kernel configuration in AADL language [6] with code generation and a possibility to represent the configuration graphically;
- it is a good *proof-of-concept* with a set of working models and examples;
- partially conforms to the *ARINC 653* real-time onboard aviation system standard [7];
- *protected partitions* with time and memory space resources isolation;
- real-time processes schedulers with different *strategies of two types*: (a) partition planner (b) process planner in each partition;
- *controllable* port and message interactions between processes; also the BlackBoard concept [7] is used.

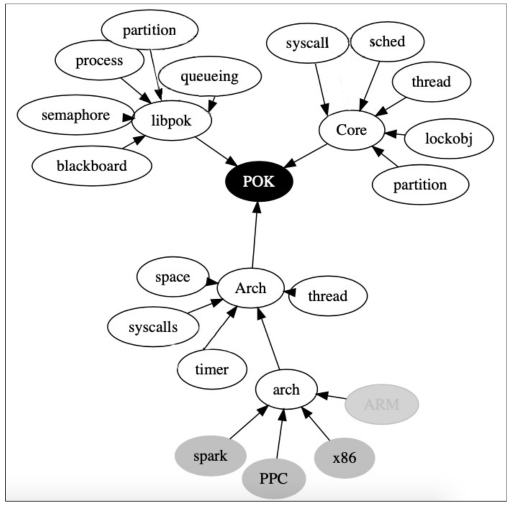


Fig. 1. A scheme of POK internal architecture

The use of OS, which is designed according to avionics standards and provides the isolation and verifiable interprocess communication, increases the robustness of the functionality of a cyber-physical system at the system level.

By browsing the source code [2], we created a scheme of internal POK architecture, shown in fig. 1. It comprises three principal layers:

- *Arch* with platform-dependent code (open-source repository includes realization for three platforms: x86-qemu, PowerPC and Sparc), also there are some works on an ARM port;
- *Core* for internal kernel code, syscalls processing;
- *libpok* can be used to call from the user's code as an API.

The ARINC 653-compatible API offers to work with partitions, processes, locking objects, ports, queries and messages in a standardized and certifiable way. The API is a high-level abstraction, in this paper, we do not touch it, and we proceed to model low-level things on which it is all based.

2.2. Model Checking with the SPIN tool

SPIN [8, 9] is a utility for verifying the correctness of distributed software models. The abbreviation SPIN stands for Simple Promela INterpreter. The SPIN system checks not the programs themselves, but their models. To build a model for an original parallel program or an algorithm, the verifying engineer (usually manually) creates a representation of this program in the C-like input language, called Promela (PROtocol MEta-LAnguage) [10].

To deal with the problem we are formalizing, we may rely upon the following language features [11]:

- it is an actor-based (process- and message-oriented) language;
- it is primarily designed to describe protocols and interoperations;
- it has C-styled syntax and fix-size finite data types;
- it uses function inlining quite similar to the macros in C;
- it allows custom types definition (using typedef as similar in C);
- it introduces "atomic" sections to model code that is running in parallel without any context switching inside.

There are no pointers, so special techniques should be used here to provide abstractions for them. For a long time, the language has been used mainly in academia, but instantly, the language authors added syntax constructions to describe complicated programs, the project has moved to GitHub and modern modular text editors (like Visual Studio Code) introduced support to highlight, refactor and run programs in Promela.

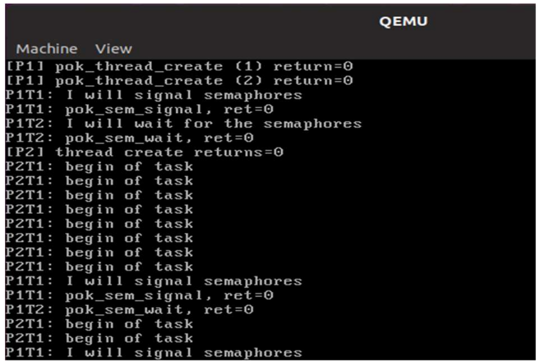
As a result, we think that it is a suitable language to model OS internals with the aim of further formal verification. Promela constructs are simple, they have clear and distinct semantics, which allows the verifier to translate any program in this language into a verifiable transition system with a finite number of states. The requirements for the model are expressed in LTL (Linear-time Temporal Logic) [12].

The model checking process inside comprises (a) converting a model program into a Büchi automaton by considering the change in its state, (b) resolving non-determinism, (c) modeling context-switching as the creation of variants of possible transitions, (d) converting the negation of a temporal formula of a requirement into an automaton, and (e) creating the resulting parallel composition of automata [11]. During the verification process, a traversing is made through all the states of the resulting automaton, plus at the same time, violations of requirements are checked as generated asserts. If the requirement is violated, the verifier produces a counterexample as a sequence of control states of the system (a trail) which points to the violation of the requirement.

3. OS Internals Modeling: Our Approach

3.1. The Sample to Study

To model the partitioned OS, we carefully studied an example with multi-threaded work of processes (located at examples/semaphores in [2], see fig. 2). We setup a C development environment with prescribed source and include paths. We walked through the source code and inspected all called functions or macros. This made it possible to recreate the behavior of a real OS.



```
[P1] pok_thread_create (1) return=0
[P1] pok_thread_create (2) return=0
P1T1: I will signal semaphores
P1T1: pok_sem_signal, ret=0
P1T2: I will wait for the semaphores
P1T2: pok_sem_wait, ret=0
[P2] thread create returns=0
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P2T1: begin of task
P1T1: I will signal semaphores
P1T1: pok_sem_signal, ret=0
P1T2: pok_sem_wait, ret=0
P2T1: begin of task
P2T1: begin of task
P1T1: I will signal semaphores
```

Fig. 2. Minimal partitioned code example from [2], working in QEMU environment.

In Listing 1, we show part of the source activity code of the Thread 1, working in the Partition 1.

```
void* pinger_job () {
   pok_ret_t ret;
   while (1) {
      printf ("P1T1: I will signal semaphores\n");
      ret = pok_sem_signal (sid);
      printf ("P1T1: pok_sem_signal, ret=%d\n", ret);
      pok_thread_sleep (2000000);
   }
}
```

Listing 1. A multi-threaded sample

In the code, one thread signals a semaphore and sleeps, and continues to do it forever. The other threads at the same time wait for the semaphore and do some work.

We choose this sample because of two things:

- it is really a minimal behavior of a partitioned OS;
- it contains multi threads, multi partitions as well as locking primitives and sleeping, so it is suitable to model dynamic scheduling algorithms.

3.2. Modeling the Activity Code in Promela

In Listing 2, we present a model for the above code.

```
proctype threadP1T1(short myPartId; short myThreadId) {
do
::(osLive == 1) ->
atomic {
 if ::(currentPartition == myPartId
  && currentThread == myThreadId && currentContext.IP == 0) ->
      {
       pok_print(P1T1_I_will_signal_semaphores);
       currentContext.IP++;
      }
   ::else ->
  if ::(currentPartition == myPartId &&
 currentThread == myThreadId && currentContext.IP == 1) ->
      {
       pok_sem_signal(sid, currentContext.r0);
       currentContext.IP++;
      }
  ::else ->
 if ::(currentPartition == myPartId &&
  currentThread == myThreadId && currentContext.IP == 2) ->
     {
      pok_printf(P1T1_pok_sem_signal_ret, currentContext.r0);
      currentContext.IP++;
     }
  ::else ->
 if ::(currentPartition == myPartId &&
  currentThread == myThreadId && currentContext.IP == 3) ->
     {
      pok_delay(2000);
      currentContext.IP = 0; /* inf loop */
     }
  ::else -> skip;
   fi
  fi
 fi
fi
}
::else -> break;
od
}
```

Listing 2. Model for the multi-threaded sample

Here we see a state machine that makes transitions between its states. A state of the process is characterized by the *IP* (instruction pointer) register. There are also the guard conditions to check if we are the current one to execute. The main idea here: *all the processes are traversing thought their states if they are active, and the OS scheduler is activated periodically and selects a current partition*

as well as a current process (changes the currentPartition and currentThread variables), and that causes the whole system model to run.

Also, there are *pok_print*, *pok_delay*, *pok_sem_signal* macros that emulate the syscalls in the OS, we consider them in the appropriate section.

As a result of the current section, we can state that any code that models some actions in a real OS must satisfy the following properties:

- for each thread in the system, a corresponding *process* is created in Promela;
- for all his calculations, it uses *only register variables* from the current context;
- after each line of significant code, the register *IP is incremented*;
- each line of code is executed *in the switch* by IP, current process and current partition.

3.3. Data Definition in the Model

Thanks to the support of *typedef* complex structures and arrays in Promela, we can build mostly a normal data definition in our model (see fig. 3). Here we introduce *Context*, *Thread*, *Partition* and *Semaphore* structures to model corresponding OS entities.

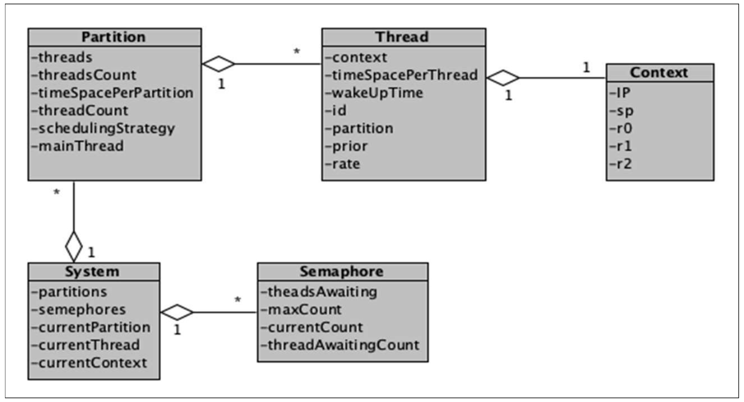


Fig. 3. Data structures in our model

To simulate the code execution, we explicitly introduce the processor registers as Promela variables and put them into the current execution context, which simulates one processor with its memory. Those are primarily a register for the current instruction pointer (*IP*), a stack register (*sp*) and several arithmetic registers (r_0-r_n), see Listing 3.

```
typedef Context {
    int IP;         //instruction pointer
    int sp;         //stack pointer - for further modeling
    int r0;         //arithmetic registers
    int r1;
    int r2;
}
```

Listing 3. Model for the state of the current thread

IP is used for the program flow in a thread (see Listing 2), arithmetic registers should be used in calculations, *sp* is added for future use (for example, to model local memory, procedures and parameter passing). Then we include such a context to the thread definition (see Listing 4).

```
typedef Thread {
    Context context;  //thread context to save
    short timeSpacePerThread; //count of ticks to run
    bit isLocked;     //1 if it has been locked on a semaphore
    int wakeUpTime;   //wake up time to schedule using 'sleep'
    short id;         //unique thread id
    short partition;  //number of the parent partition
    short prior;      //for further model with priorities
    short rate;       //current execution time - for rms
}
```

Listing 4. Data definition for threads

A thread is characterized by its context, some parameters and time space (amount of time to run the thread before the switch). Now and after we are going to count *time* in ticks, countable by the scheduler. After all, we introduce the partition definition as shown in Listing 5.

```
typedef Partition {
    short timeSpacePerPartition; //count of ticks to run
    short threadCount;
    Thread threads[MAXTHREADS];  //threads of this partition
    short schedulingStrategy;    //type of sched for threads
    short mainThread;            //first thread to run
}
```

Listing 5. Data definition for partitions

It consists of a number of threads, time space for the partition to run between a switch, scheduling strategy of related threads and the main thread to peak at first.

4. Modeling the Scheduler

For the first iteration, we show a simple non-deterministic scheduler that randomly selects a partition of two and a thread of two inside, see Listing 6.

```
proctype schedNonDeterministicInstance() {
 do
 :: realTime < MAXTIMESIM -> {
  atomic {
    saveCurrentContext();

    //non-deterministic partitions scheduler
    if
      ::true -> currentPartition = 0;
      ::true -> currentPartition = 1;
    fi

    if
      ::(currentThread == 0) -> currentThread = 1; //stub
      :: else -> currentThread = 0;
    fi
      realTime++;
      restoreCurrentContext();
   }
  }
 :: else -> {
      printf("Simulation time is over!\n");
      osLive = 0;
      break;
    }
 od
}
```

Listing 6. Simple scheduler that peaks random partitions and threads

The scheduler runs as a Promela process; it activates at some random time. The system runtime is bounded to a constant, and the *realTime* variable is used to count time passed in the whole system, so we are counting time right once the scheduler is activated (corresponds to the hardware timer

interrupt handling). The s*aveCurrentContext* and r*estoreCurrentContext* macros are used to save the current context to a place of the context of a current thread and restore it respectively. So, using ideas in Listing 2 and Listing 6, one can implement a very simple model of the scheduling.

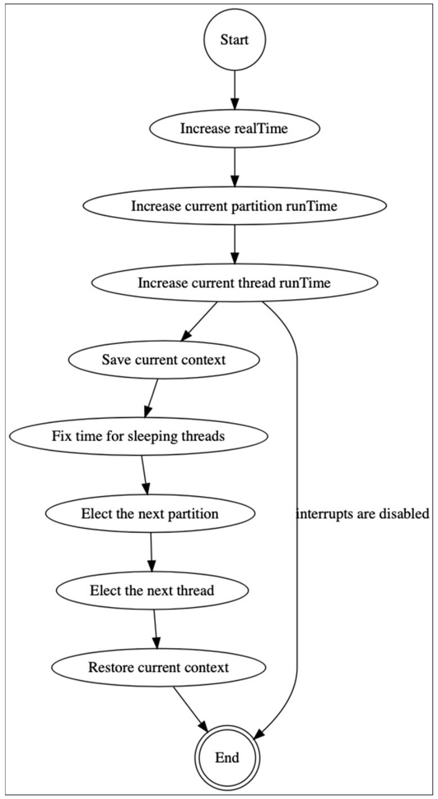


Fig. 4. A model of a partitioned scheduler

The real scheduler that we use in our model is much more sophisticated. In fig. 4 we depicted its block scheme. It runs in a loop that is fired on don-deterministic times. The first thing to do is to increment the time variables of the whole system as well as time running of a current partition and a current thread (remember, we have bounds for these times in the thread and partition definition structures). Then only if a logical variable for disabled interrupts is not set (corresponds to disabling the interrupts in the real OS), we continue to the switching process. The next thing to do — is to fix wakeup time for all the sleeping threads. That means that for all threads with elapsed time of sleeping we should remove their sleeping statuses (because we had already changed current time and some threads have just become candidates to switch to). The resting behavior of the scheduler is the same as the previous one: save current context, elect a partition, elect a thread and restore the context. However, here we do the elections according to set election strategies and current locking statuses.

In Listing 7, we show a piece of code to elect the next thread. Here we introduce scheduling strategies that are set in the partitions during the initialization phase. Then the right strategy can be applied in the scheduling loop.

```
mtype = {sched_part_rms_strategy, sched_part_rr_strategy,
sched_part_edf_strategy, sched_part_llf_strategy}
```

```
inline elect_next_thread(needPeakAThread) {
if
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_rms_strategy) -> sched_part_rms(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_rr_strategy)  -> sched_part_rr(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_llf_strategy) -> sched_part_llf(needPeakAThread);
::(partitions[currentPartition].schedulingStrategy ==
 sched_part_edf_strategy) -> sched_part_edf(needPeakAThread);
::else -> skip;
fi
}
```

Listing 7. An extensible thread election.

In Fig. 5 we show our implementation of the round-robin thread election.

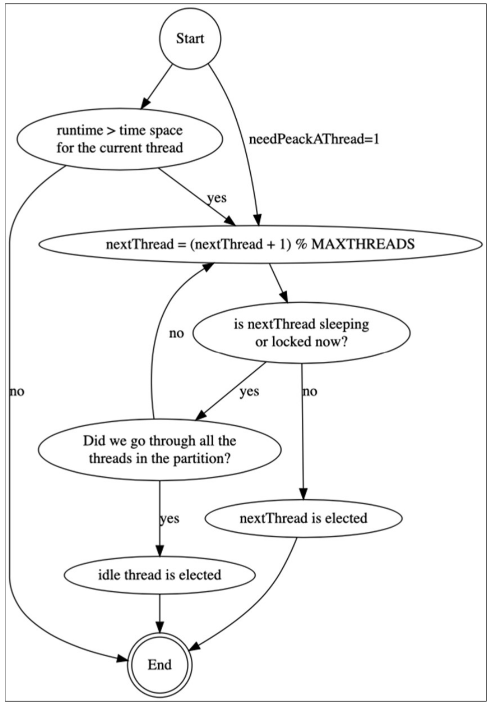

Start
runtime > time space for the current thread
needPeackAThread=1
yes
nextThread = (nextThread + 1) % MAXTHREADS
no
is nextThread sleeping or locked now?
no
yes
no
Did we go through all the threads in the partition?
yes
nextThread is elected
idle thread is elected
End

Fig. 5. A model of a round-robin scheduler with the possibility of locking and sleeping

In a loop, we select a next thread and check whether it is runnable (it means it can sleep after requesting to sleep in the code or to be locked on a semaphore). If we are not able to select the next thread, we select the virtual one (idle thread) with no code to execute. The guard conditions to start the elections are (a) the current thread has run out of its time or (b) the scheduler is asked to select a new thread (due to sleep or locking have queued).

To select a new partition, we only use the condition that the current partition has run out of its time.

5. Modeling the SysCalls

In POK, all API that OS provides to its client processes (for example, creating a semaphore, waiting for it or blocking it) are done through system calls. This means that for each interaction with a kernel object, the generation of a software interrupt is performed. This approach allows to control such calls from the OS, to be able to prioritize them, to perform them in a protected context.

During the modeling, we create an enumeration of possible syscalls, available to the user. Then we create macros to wrap API calls in a syscall executor routine (it fully compliments to the POK code). The executor prepares the syscall parameters in registers and generates a software interrupt. We model it as a message passing to a Promela channel. The syscall executor puts a signal to the channel and the syscall handler waits here for the signal in a loop, awakens and actually performs the call. In Listing 8 we demonstrate the user syscall library (see also Listing 1 for the example of its utilization).

```
//syscalls types
mtype = {syscall_sem_p, syscall_sem_v, syscall_delay,
syscall_printf}

//library available to user
inline pok_sem_signal(sid, ret) {
    printf("pok_sem_signal\n");
    pok_do_syscall(syscall_sem_v, sid, NOPARAM, ret);
}

inline pok_sem_wait(sid, ret) {
    printf("pok_sem_wait\n");
    pok_do_syscall(syscall_sem_p, sid, NOPARAM, ret);
}
```

Listing 8. Syscalls user library

In Listing 9 we show the model of syscalls executor.

```
inline pok_do_syscall(N, param1, param2, ret) {
  atomic {
    //pass the params
    currentContext.r0 = N;
    currentContext.r1 = param1;
    if
        ::(param2 != NOPARAM) -> currentContext.r2 = param2;
        ::else -> skip
    fi
    }
    //emit the interrupt
    InterruptController ! POK_INTERRUPT;
    //wait for iret
    InterruptRet ? ret;
}
```

Listing 9. Syscalls executor

In Listing 10, we show part of the model of syscalls handler with a switch by a syscall id.

```
interruptsDisabled = 1; //stop the scheduler (soft model)
saveCurrentContext();
if
 ::(intNum == POK_INTERRUPT) -> {
   if
     ::(id == syscall_sem_v)  -> sem_signal(param1);
     ::(id == syscall_sem_p)  -> sem_wait(param1);
     ::(id == syscall_delay)  -> sleep(param1);
     ::(id == syscall_printf) -> print(param1, param2);
     ::else -> skip; //unknown syscall id
```

```
    fi
  }
  ::else -> skip;
fi
restoreCurrentContext();
```

Listing 10. A part of syscalls handling

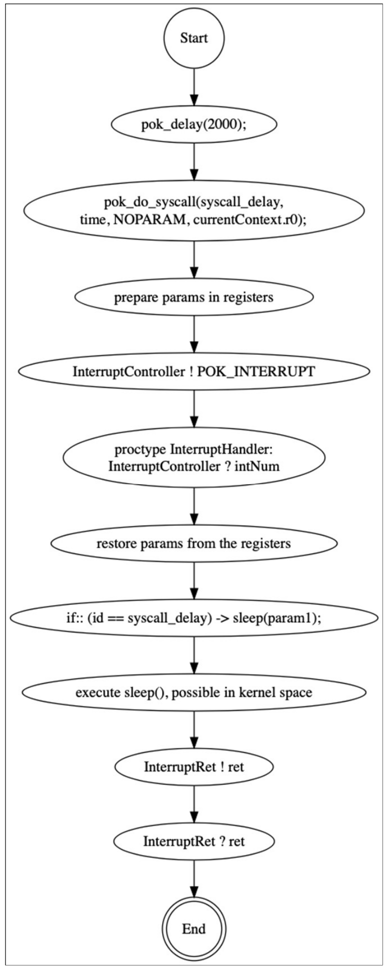

Start
pok_delay(2000);
pok_do_syscall(syscall_delay, time, NOPARAM, currentContext.r0);
prepare params in registers
InterruptController ! POK_INTERRUPT
proctype InterruptHandler: InterruptController ? intNum
restore params from the registers
if:: (id == syscall_delay) -> sleep(param1);
execute sleep(), possible in kernel space
InterruptRet ! ret
InterruptRet ? ret
End

Fig. 6. Syscalls processing model

The handler gets the parameters from the registers and uses the switch operator to decide which kernel function it should execute. The overall scheme of interrupts modeling is presented in fig. 6.

In addition, we focus on the implementation of our syscalls:

- *sleep* is implemented using the calculation of a wakeup time for the current thread based on the given delay value and call the scheduler;
- *wait on a semaphore* is implemented by updating the semaphore counter and adding the current thread to a list of awaiters of the given semaphore if it is necessary;

- *signal a semaphore* is implemented by updating the semaphore counter and removing the current thread from a list of awaiters of the given semaphore if it is necessary;
- *print* is implemented using switching by the parameter and printing a corresponding string.

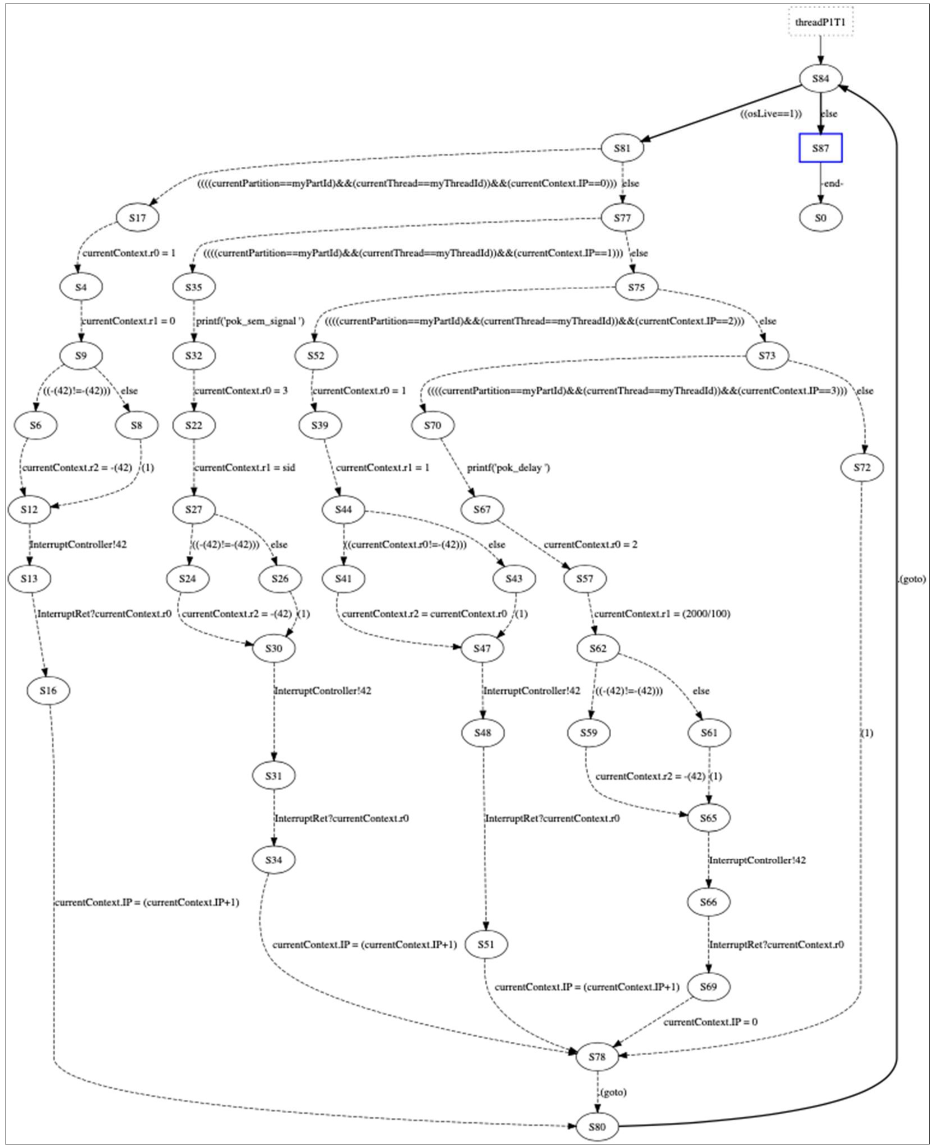

threadP1T1
S84
((osLive==1))
else
S81
S87
-end-
S0
((((currentPartition==myPartId)&&(currentThread==myThreadId))&&(currentContext.IP==0)))
else
S17
S77
currentContext.r0 = 1
((((currentPartition==myPartId)&&(currentThread==myThreadId))&&(currentContext.IP==1)))
else
S4
S35
S75
currentContext.r1 = 0
printf('pok_sem_signal ')
((((currentPartition==myPartId)&&(currentThread==myThreadId))&&(currentContext.IP==2)))
else
S9
S32
S52
S73
((-(42)!=-(42)))
else
currentContext.r0 = 3
currentContext.r0 = 1
((((currentPartition==myPartId)&&(currentThread==myThreadId))&&(currentContext.IP==3)))
else
S6
S8
S22
S39
S70
S72
currentContext.r2 = -(42)
(1)
currentContext.r1 = sid
currentContext.r1 = 1
printf('pok_delay ')
S12
S27
S44
S67
InterruptController!42
((-(42)!=-(42)))
else
((currentContext.r0!=-(42)))
else
currentContext.r0 = 2
S13
S24
S26
S41
S43
S57
(goto)
InterruptRet?currentContext.r0
currentContext.r2 = -(42)
(1)
currentContext.r2 = currentContext.r0
(1)
currentContext.r1 = (2000/100)
S30
S47
S62
S16
InterruptController!42
InterruptController!42
((-(42)!=-(42)))
else
S48
S59
S61
(1)
S31
currentContext.r2 = -(42)
(1)
InterruptRet?currentContext.r0
InterruptRet?currentContext.r0
S65
S34
InterruptController!42
S66
currentContext.IP = (currentContext.IP+1)
currentContext.IP = (currentContext.IP+1)
S51
InterruptRet?currentContext.r0
currentContext.IP = (currentContext.IP+1)
S69
currentContext.IP = 0
S78
(goto)
S80

Fig. 7. Automaton representation of a process in our OS model.

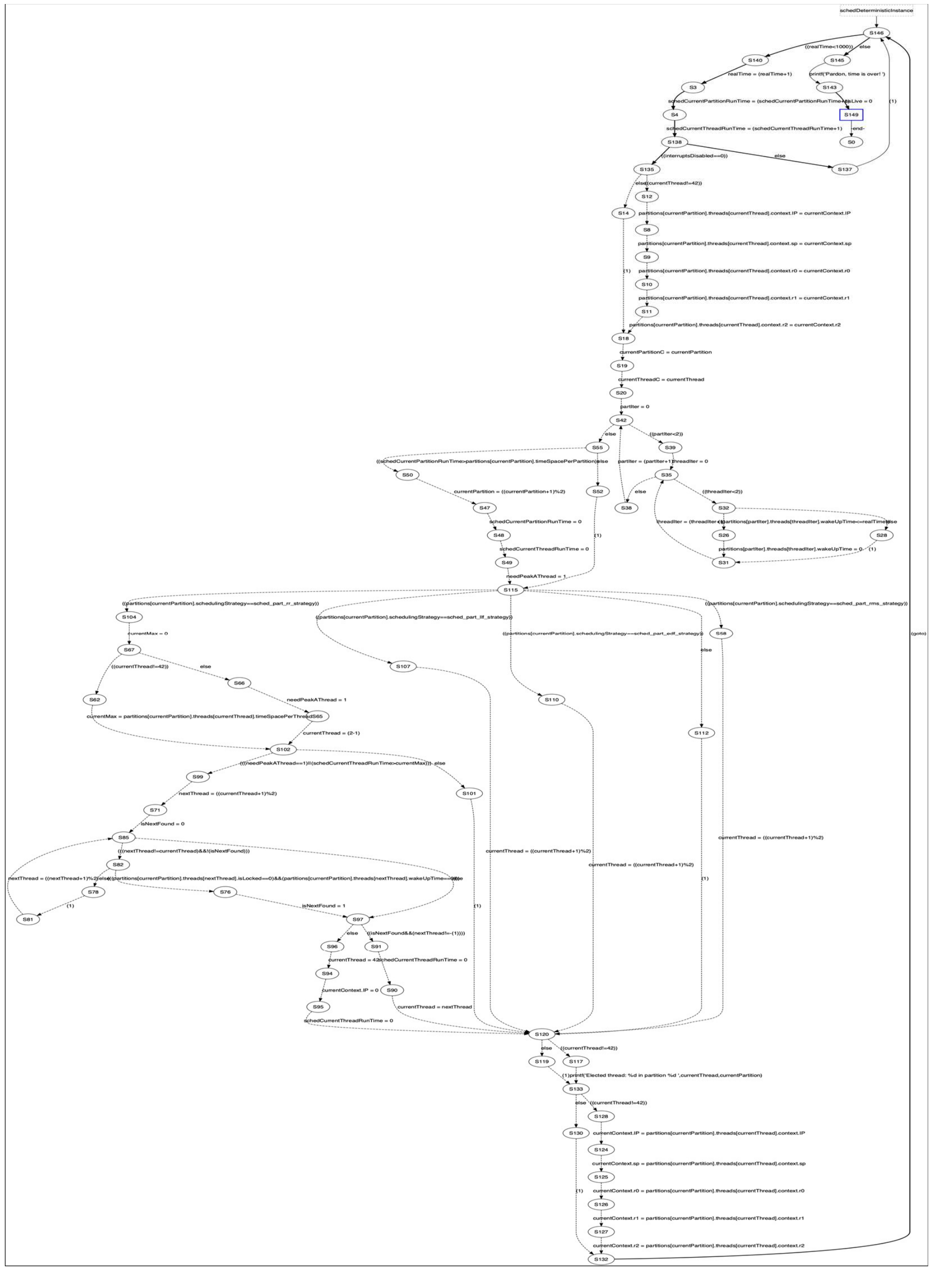


Fig. 8. Automaton representation of the scheduler in our OS model.

6. Related work

Today's OS for reliable cyber-physical systems like flight machines should be definitely real-time and must offer memory space and time division capabilities. There exists the avionics industry-related standard for these requirements, created by the Aeronautical Radio, Inc., ARINC 653 [7].

Methods of validation of ARINC architectures were given in [13]. Some techniques on verification of OS (mostly related to Linux) were presented by the ISP RAS in [14-16].

In Russia, a certified partitioned operating system intended for the aircraft, was created by GosNIIAS and ISP RAS with advanced debugging capabilities, rewritten scheduler, system partition feature and different platforms support [17, 18], some results were GPLv3 licensed.

The most famous approach to verification of OS is presented in [19]. The authors created executable specifications of an L4 microkernel in Haskell based on initial C implementation and then refined them into an Isabelle/HOL model. There is also a good literature review on this area in their paper. Our approach follows theirs: we created an executable specification in Promela according to the C code of POK, but then we are going to use model checking methods instead of theorem proving.

7. Discussions

7.1. Visualization of the Model

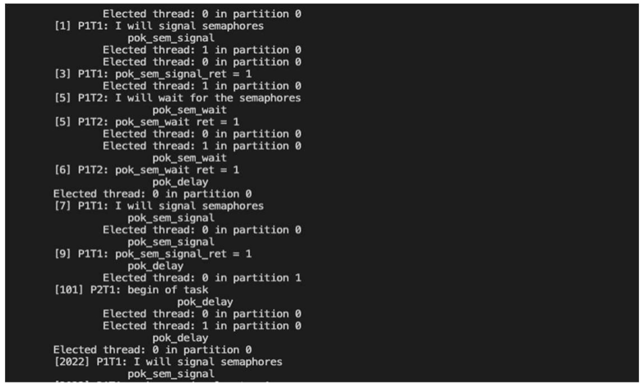
Using the SPIN capability to export model automata as .dot diagrams (*./pan -D* [11]), we created the automata representations of a process (see fig.7) as well as of the scheduler (see fig.8). These images are presented here to estimate the complexity of resulting automata.

The process automation (corresponds to Listing 1) consists of about 80 states, we can see that some states are duplicated due to inline macros to execute the syscalls (see Section 5). The scheduler automaton consists of about 150 states.

Building such automata by hand is very costly, so the executable specification in Promela really helps to obtain a formal model to provide further checks. In addition, the model is very extensible, it is easy to add scheduling strategies (see Listing 7 for the reference) as well as implement additional syscall types, etc.

7.2. Simulation of the Model

In fig. 9 we depict a simulation process of our model, using the command-line SPIN run.



```
                Elected thread: 0 in partition 0
      [1] P1T1: I will signal semaphores
                pok_sem_signal
                Elected thread: 1 in partition 0
                Elected thread: 0 in partition 0
      [3] P1T1: pok_sem_signal_ret = 1
                Elected thread: 1 in partition 0
      [5] P1T2: I will wait for the semaphores
                    pok_sem_wait
      [5] P1T2: pok_sem_wait ret = 1
                Elected thread: 0 in partition 0
                Elected thread: 1 in partition 0
                    pok_sem_wait
      [6] P1T2: pok_sem_wait ret = 1
                pok_delay
      Elected thread: 0 in partition 0
      [7] P1T1: I will signal semaphores
                pok_sem_signal
                Elected thread: 0 in partition 0
                pok_sem_signal
      [9] P1T1: pok_sem_signal_ret = 1
                pok_delay
                Elected thread: 0 in partition 1
      [101] P2T1: begin of task
                    pok_delay
                Elected thread: 0 in partition 0
                Elected thread: 1 in partition 0
                    pok_delay
      Elected thread: 0 in partition 0
      [2022] P1T1: I will signal semaphores
                pok_sem_signal
```

Fig. 9. Model simulation using SPIN

We see that the processes work as expected (we apply the soft real-time strategy here), the scheduler does partitions as well as threads switching, and the processes wait expected time then do the interprocess communications using the semaphore.

7.3. On the Model Verification

Although we are still thinking about properties for verification, in this subsection we provide a way to check the correctness of partitions switching. According to our model, we use some string constants (see the parameters to *pok_print* in Listing 1). Since we know which partition each string belongs to, we can set the expected match of each string to its partition, see Listing 11.

```
//string constants
#define P1T1_I_will_signal_semaphores 0
#define P1T1_pok_sem_signal_ret 1
#define P1T2_I_will_wait_for_the_semaphores 2
#define P1T2_pok_sem_wait_ret 3
#define P2T1_begin_of_task 4

//map data-partition
short partitionByDataIndex[5] = {
        PARTITION1,
        PARTITION1,
        PARTITION1,
        PARTITION1,
        PARTITION2
};
```

Listing 11. Output strings and a mapping to the partitions

Then, as we know the expected partition matching and will know the actual one when we run the model, we provide the following check macro, see Listing 12.

```
//check if we are in the correct partition
inline checkPointer(expectedPartition, actualPartition) {
    if
        :: (expectedPartition != actualPartition) ->  {
            pointersOk = 0;
            printf("segmentation fault!\n");
        }
        :: else -> skip
    fi
}
```

Listing 12. A macro to check the safety of data strings

And the macro *checkPointer* is now used in the implementation of *print* syscall, see Listing 13.

```
inline print(string, param) {
 checkPointer(partitionByDataIndex[string], currentPartition);
 if
  ::(string == P1T1_I_will_signal_semaphores) ->
    printf("[%d] P1T1: I will signal semaphores\n", realTime);
  ::(string == P1T1_pok_sem_signal_ret) ->
  printf("[%d] P1T1: pok_sem_signal_ret = %d\n", realTime, param);
   ...
  ::else -> skip
 fi
}
```

Listing 13. A fragment of print syscall implementation

Therefore, the verification process will be checking an LTL formula «*it is always that pointers are ok*»:

$$\boldsymbol{G}\ (pointersOk\ ) \tag{1}$$

where *pointersOk* variable can be changed when a string is requested from an incorrect partition (see Listing 12) during all possible runs of the model.

8. Conclusion

In this paper, we analyzed the purpose, composition, and structure of the partitioned real-time OS. We discussed a possible approach for creating a model of such an OS in Promela. The proposed solution allows us to move from complex architecturally dependent code in C to general operating system behavioral models that can be used in the education process. Also, having a formalized OS model, we can check the security properties of code execution in partitioned systems with a possibility to apply different scheduling algorithms.

Possible future steps are:

- modeling of different scheduling strategies;
- modeling of ARINC API [7];
- creation of control variables, construction of LTL formulas and model verification;
- multicore scheduling models;
- models both for the hard real-time as well as the soft real-time using scheduling strategies;
- checking of cyber-physical models running in such an OS using our library [20].

The current code for the implementation of the approach described in this paper is freely available in [21]. The authorship is registered in the database of the Federal Institute of Industrial Property [22].

References / Список литературы

[1]. Staroletov S.M., Amosov M.S., Shulga K.M. Designing robust quadcopter software based on a real-time partitioned operating system and formal verification techniques. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 39-60. DOI: 10.15514/ISPRAS-2019-31(4)-3.
[2]. POK kernel repository. Available from: https://github.com/pok-kernel/pok, accessed 10.11.2020.
[3]. Basic Spin Manual. Available from: http://spinroot.com/spin/Man/Manual.html, accessed 10.11.2020.
[4]. Delange J. Intégration de la sécurité et de la sûreté de fonctionnement dans la construction d'intergiciels critiques. THÈSE, Télécom ParisTech, 2010 (in French).
[5]. Delange J., Gilles O., Hugues J., and Pautet L. Model-based engineering for the development of ARINC653 architectures. SAE International Journal of Aerospace, vol. 3, issue 1, 2009, pp. 79-86.
[6]. Delange J. AADL in Practice: Become an expert in software architecture modeling and analysis. Reblochon Development Company, 2017, 252 p.
[7]. Specification ARINC. 653-2: Avionics application software standard interface: Part 1-required services. Technical report, Avionics Electronic Engineering Committee (ARINC), 2006.
[8]. Holzmann G. The model checker SPIN. In IEEE Transactions on software engineering, vol. 23, issue 5, 1997, pp. 279-295.
[9]. SPIN. Available from: https://github.com/nimble-code/Spin, accessed 10.11.2020.
[10]. Promela language reference. Available from: http://spinroot.com/spin/Man/promela.html, accessed 10.11.2020.
[11]. Старолетов С.М. Основы тестирования и верификации программного обеспечения. СПб., Лань, 2018 г., 344 стр. / Staroletov S. Basics of Verification and Testing. Spb., Lanbook, 2018, 344 p. (in Russuan).
[12]. Pnueli A. The temporal logic of programs. In Proc. of the 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), 1977, pp. 46-57.
[13]. Hugues J., Delange J. Model-based design and automated validation of ARINC653 architectures using the AADL. In Cyber-Physical System Design from an Architecture Analysis Viewpoint, Springer, 2017, pp. 33-52.
[14]. Khoroshilov A. On formalization of operating systems behaviour verification. In Proc. of the International Conference on Computer Science and Information Technology (CSIT), 2017, pp. 168-172.
[15]. Khoroshilov A.V., Kuliamin V.V., Petrenko A.K. Verification of Operating System Components. System Informatics, issue 10, 2017, pp. 11-22.
[16]. Кулямин В.В., Лаврищева Е.М., Мутилин В.С., Петренко А.К. Верификация и анализ вариабельных операционных систем. Труды ИСП РАН, том 28, вып. 3, 2016 г., стр. 189-208 / Kuliamin V.V., Lavrischeva E.M., Mutilin V.S., Petrenko A.K. Verification and analysis of variable

operating systems. Trudy ISP RAN/Proc.ISP RAS, vol.28, issue 3, 2016, pp. 189-208 (in Russian). DOI: 10.15514/ISPRAS-2016-1(2)-12.

[17]. Mallachiev K. M., Pakulin N. V., Khoroshilov A. V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181-192. DOI: 10.15514/ISPRAS-2016-28(2)-12.

[18]. Солоделов Ю.А., Горелиц Н.К. Сертифицируемая бортовая операционная система реального времени JetOS для российских проектов воздушных судов. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 171-178 / Solodelov Y. A., Gorelits N. K. Certifiable onboard real-time operation system JetOS for Russian aircrafts design Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017, pp. 171-178. DOI: 10.15514/ISPRAS-2017-29(3)-10.

[19]. Klein G. et al. seL4: Formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, 2009, pp. 207-220.

[20]. Staroletov S., Shilov N. Applying Model Checking Approach with Floating Point Arithmetic for Verification of Air Collision Avoidance Maneuver Hybrid Model. Lecture Notes in Computer Science, vol. 11636, 2019, pp. 193-207.

[21]. Staroletov S. Partitioned OS Model Implementation. Available at: https://github.com/SergeyStaroletov/PromelaSamples/blob/master/Sched.pml, accessed 10.11.2020. DOI: 10.5281/zenodo.3757397.

[22]. Старолетов С. Модель многораздельной операционной системы для целей формальной верификации. Государственная регистрация программы для ЭВМ, no. 2020614016, 2020 / Staroletov S. Partitioned OS model for formal verification purposes. State registration of computer software, no 2020614016, 2020 (in Russian).

Information about the author / информация об авторе

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент кафедры прикладной математики. Сфера научных интересов: формальная верификация, model checking, киберфизические системы, операционные системы.

Sergey Mikhailovich STAROLETOV – Candidate of Physical-Mathematical Sciences (PhD), Associate Professor at the department of Applied Mathematics. Research interests: formal verification, model checking, cyber-physical systems, operating systems.