A formalisation of the OSEK concurrency model

Martin Schwarz

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{schwmart}@in.tum.de

1 Introduction

The OSEK[1] Operating System Specification [11] was created to increase portability of applications written for embedded systems with interrupt-driven concurrency. Its development started in 1993 as a joint effort of the European car industry and has now resulted in a partially ISO certified open standard. While a wealth of techniques are available for analysing sequential programs, much less is known about the analysis of interrupt-driven programs with priorities and task activation. Despite of the complex control-flow, programs to be executed on an OSEK-compliant system are meant to exhibit an essentially sequential behaviour. The goal is to make this explicit and exploit it for transferring techniques for the analysis of sequential programs to OSEK programs.

The OSEK specification defines a unified operating system (OSEK OS), which executes the tasks and interrupts of OSEK programs in a well-defined manner and provides a set of library functions for resource management and scheduling. We shall for convenience consider interrupts as a subset of tasks. Programs written for an OSEK OS consist of two parts: a static description file and C-files containing the actual code. In the description file the structure of the program is outlined and the attributes of each task are defined. Examples of information provided there can be seen in the code snippet in Figure 1 and include the priorities of tasks, the autostart flag which determines whether the program starts execution with the given task ready for execution, the number of instances of a task that can exist simultaneously, and the sets of accessible resources for each task. Interrupts have a more fixed behaviour, e.g. can't autostart, limited to one instance, and therefore define less attributes. We assume this information to be accessible to us through functions generated via preprocessing.

The basic scheduling policy of OSEK OS is to work through the activated tasks in priority order. Once started, a task will run to termination unless a task of higher priority is activated and pre-empts it, i.e. no time-slicing is used. This property allows the translation of OSEK programs to a sequential, stack-based execution model.

For synchronisation the Priority Ceiling Protocol (PCP) is used. The key idea of the PCP is to raise the priority of a task which has acquired a resource to the highest priority of all tasks declared to use that resource. We refer to

[1] "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug"

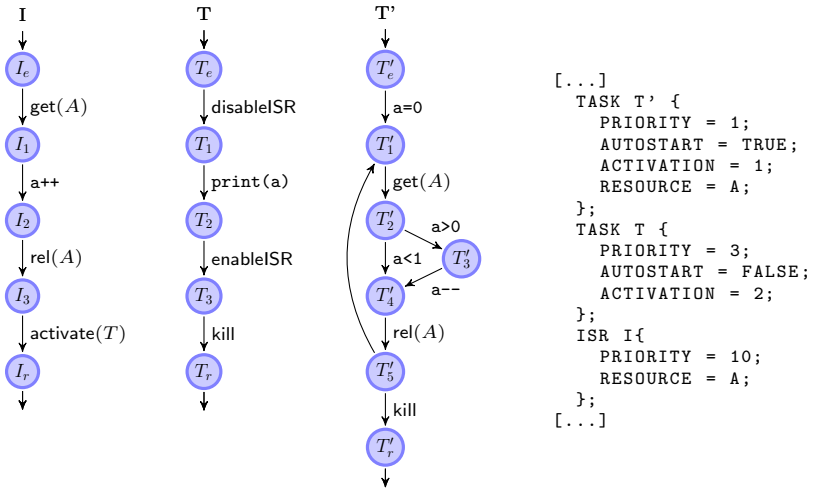

Fig. 1. Example program.

this priority as the *ceiling priority* of the resource. This policy ensures dead-lock freedom and minimises priority inversion, where a high-priority task waits for a lower-priority task to release a resource.

The program in Figure 1, an instance of a consumer-producer scheme, will serve as a running example. The program variable `a` is a counter of items available for consumption. The task T' consumes an item (not shown) and decreases the counter if there is something to be consumed; otherwise, it waits until the next item arrives. The ISR I is triggered as soon as a new item is produced. It increases the counter `a` and activates the printing task T. This small program already shows some of the interesting consequences of the PCP. As T has a higher priority than T' the printed output will always be at least one because T' will only be scheduled to resume after T has terminated.

The priorities place restrictions on the execution, ruling out many interleavings which otherwise would constitute a data race. For example, while holding the resource A, the task T' has a ceiling priority of 10 because the highest priority task declared to use the resource is I, which has a static priority of 10. Therefore, T' cannot be interrupted by I when holding this resource. Note that as long as the resource A is assigned to I in the description file, there is no data race even when the resource acquisition in the body of I is removed. This is because the interrupt cannot be pre-empted by lower-priority tasks and the ceiling priority of T' when it acquires the resource A is still 10, which ensures that it cannot be pre-empted by I. Only if we also remove the declaration that I may acquire A from the description file, can the task T' be interrupted as acquiring the resource no longer raises its priority. As will be seen in Section **??**, our analysis is sensitive to these distinction. We here provide a brief glimpse of our approach. To verify the absence of races in the program, we compute the

following information for the nodes where data is accessed:

Node	Task	Resource	Queue	Offense	Defense
I_1	I	$\{A\}$	$\{\emptyset\}$	10	10
T_1	T	$\{ISR\}$	$\{\emptyset, \{T\}\}$	3	10
T_3'	T'	$\{A\}$	$\{\emptyset\}$	1	10

As there is no pair of accesses to the variable a where one access has a strictly higher *offensive* priority than the *defensive* priority of the other, we conclude that the program is free from races.

2 Semantics of OSEK Programs

This section explains the principles of the OSEK OS and the PCP. Simultaneously we introduce a small step operational semantics which captures these principles. More precisely, we model the behaviour of systems which conform to the BCC2 conformance class of the OSEK specification. This excludes event-based task communication, the precise analysis of which is undecidable. For most OSEK library functions we use shortened versions of the OSEK names, replacing GetResource() with get(), for example.

Formally we notate the set of all tasks as Task and the set of all resources as Res. Each task $q \in \mathsf{Task}$ is represented as a *control flow graph* (CFG) as in Figure 1. A control flow graph $G_q = (N_q, E_q, q_e, q_r)$ consists of a set N_q of *program points*; a set of edges $E_q \subseteq N_q \times \mathsf{Stmt} \times N_q$ annotated with commands; a special *entry point* $q_e \in N_q$; and a special *return point* $q_e \in N_q$. We assume here that the program points of different procedures are disjoint, and denote by N and E the sets of all nodes and edges, respectively.

Since there is no *main*-task in an OSEK program, we construct an artificial *main*-task which consists of one node $main_e$ and one edge $(main_e, \mathsf{schedule}, main_e)$. As actual priorities are positive we set -1 as the priority of the main task which ensures that it does not interfere wiht the actual program. Execution starts at $main_e$, with no resources held and all tasks ready which according to the OIL-file, possess the AUTOSTART attribute. If there are no auto-starting tasks, nothing happens until an interrupt occurs.

To handle priorities we compute a function $\mathcal{P} : 2^{\mathsf{Res}} \to \mathbb{N}$ that maps sets of resources to the maximum of ceiling priorities of resources in the set. In the example $\mathcal{P}(A) = 10$.

During the execution of a OSEK program, the tasks move between the states *suspended*, *ready* or *running*. At most one task is *running* and when a task terminates it becomes *suspended*. In our semantics the current node is *running*. *Ready* tasks are kept in a priority queue Q which we represent as a list. Since the priority depends on the set of held resources, it is stored along with the node. The function $hd(\mathrm{Q})$ yields the first, highest priority, element and $tl(\mathrm{Q})$ yields the queue without the head. In case the queue is empty hd returns $\langle main, \emptyset \rangle$.

The auxiliary functions *add* and *push* insert an element into the queue as the newest or oldest element of its priority, respectively. Since the amount of

instances of a task q is limited, we write this number as $|q|$, $add(u, \mathrm{Q})$ may return Q unchanged if $u \in N_q$ and $|q|$ is reached. With π_i denoting the projection to the i-th component of a tuple and $\mathrm{Q}_{|q}$ the restriction of the queue to only nodes in N_q, *add* and *push* are defined as follows:

$$add(\langle u, R\rangle, \mathrm{Q}) = \begin{cases} \mathrm{Q} & u \in N_q,\ |\mathrm{Q}_{|q}| \geq |q| \\ \langle u, R\rangle : \mathrm{Q} & \mathcal{P}(R) > \mathcal{P}(\pi_2(hd(\mathrm{Q}))) \\ hd(\mathrm{Q}) : add(\langle u, R\rangle, tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

$$push(\langle u, R\rangle, \mathrm{Q}) = \begin{cases} \langle u, R\rangle : \mathrm{Q} & \mathcal{P}(R) \geq \mathcal{P}(\pi_2(hd(\mathrm{Q}))) \\ hd(\mathrm{Q}) : push(\langle u, R\rangle, tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

In our example program of Figure 1, $|T| = 2$ and $|T'| = 1$. Since I is an interrupt, $|I| = 1$. Since always one task will be running, there are at most three tasks in the queue.

According to the OSEK specification, the currently running task is counted as one active instance. Our definition of *add* as is, does not take care of this. This counting policy, though, can be simulated by pushing a copy of the running task into the queue before calling *add* and removing it again after this call.

Our semantics only considers the set of held resources to determine the priority of a task. According to the OSEK specification, though, further information must be taken into account. These are the static priorities defined in the description file; various OSEK library functions disabling interrupts as well as locking of the scheduler (viewed as an implicit resource) through get. This behaviour of the scheduler suggests to reduce all these cases to acquiring artificial resources via get. Accordingly, we create distinct artificial resources for the respective library functions, assign them to all affected tasks in the description file and replace the library function calls with get() for the corresponding artificial resources.

For example program of Figure 1, this means that the command disableISR is replaced by $\mathsf{get}(r_{ISR})$ where the new artificial resource r_{ISR} is assigned to I. The scheduler resource r_{sched} would be assigned to all tasks except interrupts.

Analogously, the static priority of a task q is represented by the artificial resource r_q which is assigned solely to q. The corresponding commands get() and rel(), we include directly in the semantics instead of the source in order to avoid spurious pre-emptions. With these artificial resources all priority information is bundled in the set of held resources.

A state of our semantics consists of the current node $u \in N$, the set of held resources $R \in 2^{\mathsf{Res}}$ of the running task, and the queue $\mathrm{Q} \in \mathsf{Queue}(N \times 2^{\mathsf{Res}})$ of ready tasks. We don't explicitly carry around components for other global or intra-task information.

For commands c which are not related to scheduling, we write $c \in$BASIC and bundle them in a single rule. Besides all C statements, BASIC contains the OSEK commands get(), rel() and activate() which operate on the state as follows:

$$[\![\mathsf{get}(r)]\!]\langle R, \mathrm{Q}\rangle = \langle R \cup \{r\}, \mathrm{Q}\rangle \qquad [\![\mathsf{rel}(r)]\!]\langle R, \mathrm{Q}\rangle = \langle R \setminus \{r\}, \mathrm{Q}\rangle$$
$$[\![\mathsf{activate}(q)]\!]\langle R, \mathrm{Q}\rangle = \langle R, add(\langle q_e, \{r_q\}\rangle, \mathrm{Q})\rangle$$

$$\frac{(u,c,v) \in E \qquad c \in \text{BASIC} \qquad \langle R', \mathrm{Q}' \rangle = [\![c]\!]\langle R, \mathrm{Q} \rangle}{\langle u, R, \mathrm{Q} \rangle \implies \langle v, R', \mathrm{Q}' \rangle} \text{ BASIC}$$

$$\frac{(u, \mathsf{schedule}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q}) \qquad \mathcal{P}(R) < \mathcal{P}(R')}{\langle u, R, \mathrm{Q} \rangle \implies \langle u', R', push(\langle v, R \rangle, tl(\mathrm{Q})) \rangle} \text{ SWITCH}$$

$$\frac{(u, \mathsf{schedule}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q}) \qquad \mathcal{P}(R) \geq \mathcal{P}(R')}{\langle u, R, \mathrm{Q} \rangle \implies \langle v, R, \mathrm{Q} \rangle} \text{ STAY}$$

$$\frac{(u, \mathsf{kill}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q})}{\langle u, R, \mathrm{Q} \rangle \implies \langle u', R', tl(\mathrm{Q}) \rangle} \text{ KILL}$$

$$\frac{q \in \mathsf{Irpt} \qquad R' = \{r_q\} \qquad \mathcal{P}(r_q) > \mathcal{P}(R)}{\langle u, R, \mathrm{Q} \rangle \Rightarrow \langle q_e, R', add(\langle u, R \rangle, \mathrm{Q}) \rangle} \text{ IRPT}$$

Fig. 2. Queue-based semantics ($\Rightarrow$).

The PCP is realised by the *scheduler* routine of the OSEK OS. The scheduler is not constantly monitoring the queue. Instead, it acts when certain conditions occur or library functions are called. Scheduling, e.g., is invoked when a resource is released via the command rel(), when readying a task via activate(), or terminating a task via kill. The scheduler can also explicitly be called via the command schedule. In the example from the previous section, scheduling may occur at the nodes T_r, T_5' and T_r'. The completion of an interrupt calls the scheduler, hence a scheduling may indirectly also occur, e.g., at node T_1', allowing T to execute before T' resumes. On the other hand, the nodes T_2' to T_4' are visited contiguously as the priority at these nodes is so high that I is not allowed to run, and the nodes T_1 and T_2 are also visited contiguously since interrupts are disabled.

In order to make all calls of the scheduler explicit, we remove all implicit triggers and instead insert the command schedule after every triggering command. This separates commands modifying the program state from commands modifying the control flow. The only exception is the command kill where the running task is replaced with the next task which the scheduler would select. This is expressed by the KILL-rule in Figure 2. This may, due to the queue being empty, result in returning to the artificial node *main* where the program would idle until an interrupt occurs.

When the scheduler is called, it selects the ready task with the highest priority from the queue, switches its state into running and starts executing it. Should several tasks have the same priority, the oldest one is selected (see Figure 3). When a tasks becomes running, it keeps its age. In case of pre-emption, it will still be the oldest among the same priority tasks. This behaviour is already captured by the functions *push* and *add* defined above. The next task to be selected from the queue Q is thus retrieved by $hd(\mathrm{Q})$.

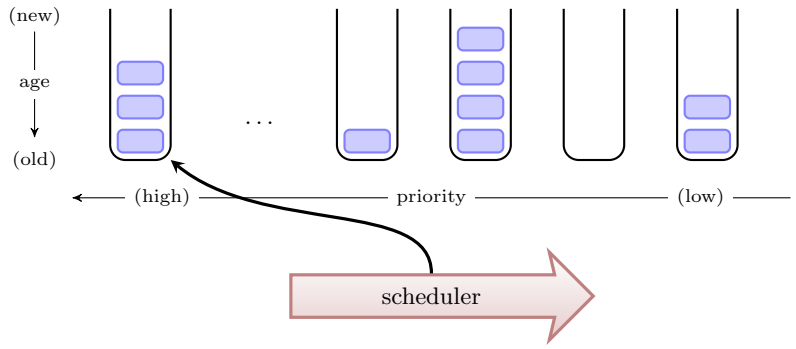

Fig. 3. Scheduling of tasks in the queues

After selecting the next task, the scheduler checks whether it has a strictly higher priority than the running task. If not, the scheduler will do nothing. See the STAY-rule. Otherwise, pre-emption occurs. This means that the current running task is stopped, its state is changed to ready and the selected task becomes running. See the SWITCH-rule.

In our semantic description, we used $[\![\bullet]\!]$ in the BASIC-rule to refer to the standard semantics of C. A specific analysis may replace this with a suitable abstraction of the semantics without interfering with the scheduling. The queue-based semantics of OSEK can be extended with possibly recursive functions. In this case, a configuration is represented as a call-stack for the running task together with a task queue which now consists of the call-stacks of fresh or pre-empted tasks.

3 Stack-Based Semantics of OSEK

In this section, we show that the queue-based small-step semantics of the last section is equivalent to a stack-based small-step semantics. There is almost a one-to-one correspondence between the rules of the semantics of Figure 2 and the rules of the new stack-based semantics as presented in Figure 4. Only the KILL-rule is split into two cases. What is different, however, are the data structures for storing ready tasks. Instead of maintaining a single queue, we maintain a stack of pre-empted tasks together with a simpler and smaller queue of names of fresh tasks only. Therefore, a call of the scheduler behaves similar to an indirect function call, where the SWITCH-rule and RESUME-rule, act as *call* and *return*, respectively. The two cases of the KILL-rule refer to resuming a pre-empted task from the stack and to starting the next task from the queue. The functions *add* and *push* still apply $\mathcal{P}$ to the resource set of their first argument. Now, however, this set consists only of the artificial resource corresponding to the static priority of the added task. This resource can be retrieved directly from the name of the task and therefore need not be stored inside the queue.

$$\frac{(u,c,v) \in E \qquad c \in \text{BASIC} \qquad \langle R', \mathrm{Q}' \rangle = [\![c]\!] \langle R, \mathrm{Q} \rangle}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R'\rangle, \mathrm{Q}'\rangle} \text{ BASIC}$$

$$\frac{(u,\mathsf{schedule},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathrm{Q}' = tl(\mathrm{Q}) \qquad \mathcal{P}(R) < \mathcal{P}(R')}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R\rangle \,;\langle hd(\mathrm{Q})_e, R'\rangle, \mathrm{Q}'\rangle} \text{ SWITCH}$$

$$\frac{(u,\mathsf{schedule},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathcal{P}(R) \geq \mathcal{P}(R')}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R\rangle, \mathrm{Q}\rangle} \text{ STAY}$$

$$\frac{(u,\mathsf{kill},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathrm{Q}' = tl(\mathrm{Q}) \qquad \mathcal{P}(\tilde{R}) < \mathcal{P}(R')}{\langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle hd(\mathrm{Q})_e, R'\rangle, \mathrm{Q}'\rangle} \text{ NEXT}$$

$$\frac{(u,\mathsf{kill},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathcal{P}(\tilde{R}) \geq \mathcal{P}(R')}{\langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle, \mathrm{Q}\rangle} \text{ RESUME}$$

$$\frac{q \in \mathsf{Irpt} \qquad \mathcal{P}(\{r_q\}) > \mathcal{P}(R)}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle u, R\rangle \,;\langle q_e, \{r_q\}\rangle, \mathrm{Q}\rangle} \text{ IRPT}$$

Fig. 4. Stack-based semantics ($\to$).

In the following, we present a translation Φ of configurations of the queue-based semantics into configurations of the stack-based semantics:

$$\Phi : N \times (2^{\mathsf{Res}} \times \mathsf{Queue}(N \times 2^{\mathsf{Res}})) \to \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Task})$$

$$\Phi\langle u, \langle R, \mathrm{Q}\rangle\rangle = \langle \phi_\Gamma(\mathrm{Q}) \,;\langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$$

Note that Task only contains *task names* so it is equivalent to the set of all entry nodes of tasks, which is a small subset of N. Assume that the configuration of the queue-based semantics is given by $\langle u, \langle R, \mathrm{Q}\rangle\rangle$. Then the function ϕ_Q removes partially executed tasks from the queue Q returning a queue of fresh tasks only and is given by:

$$\phi_Q(\mathrm{Q}) = \begin{cases} [\,] & \mathrm{Q} = [\,] \\ push(q, \phi_Q(tl(\mathrm{Q}))) & hd(\mathrm{Q}) = \langle q_E, \{r_q\}\rangle \\ \phi_Q(tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

The function ϕ_Γ builds the stack of partially executed tasks from Q. The depth of this stack is at most the number of priorities below the static priority of the currently running task. The function is defined as follows:

$$\phi_\Gamma(\mathrm{Q}) = \begin{cases} [\,] & \mathrm{Q} = [\,] \\ \phi_\Gamma(tl(\mathrm{Q})) & hd(\mathrm{Q}) = \langle q_e, \{r_q\}\rangle \\ \phi_\Gamma(tl(\mathrm{Q})) \,;\langle u, R\rangle & hd(\mathrm{Q}) = \langle u, R\rangle \end{cases}$$

Note that since the introduced artificial resources are the same in both semantics and the sets of held resources are not modified the priority function $\mathcal{P}$ commutes with the translation Φ.

Theorem 1. *For any two reachable configurations* $\langle u, \langle R, Q\rangle\rangle, \langle v, \langle R', Q'\rangle\rangle$*, the following equivalence holds:*

$$\langle u, \langle R, Q\rangle\rangle \Rightarrow \langle v, \langle R', Q'\rangle\rangle \iff \Phi\langle u, \langle R, Q\rangle\rangle \rightarrow \Phi\langle v, \langle R', Q'\rangle\rangle$$

Proof. Recall that there is a one-to-one correspondence between the rules of the queue-based and the stack-based semantics — up to the KILL-rule which is simulated by the two rules NEXT and RESUME. We perform a case distinction on the rules and prove that a rule is applicable to a configuration of the queue-based semantics if and only if the corresponding rule is applicable to the translated configuration where the results of the application are again in relation via the translation Φ. Note that we require all configurations to be reachable. i.e. proper priorities, thus we won't explicitly state it every time. Consider, e.g., the $(\Rightarrow)$-SWITCH-rule at an edge $(u, \mathsf{schedule}, v)$ and a configuration $\langle u, \langle R, \mathrm{Q}\rangle\rangle$ of the queue-based semantics. Let $\langle u', R'\rangle = hd(\mathrm{Q})$. Then the $(\Rightarrow)$-SWITCH-rule is applicable iff $\mathcal{P}(R) < \mathcal{P}(R')$, with resulting configuration $\langle u', \langle R', \mathrm{Q}'\rangle\rangle$ where $\mathrm{Q}' = push(\langle v, R\rangle, tl(\mathrm{Q}))$. On the other hand, consider the configuration $\Phi(\langle u, \langle R, \mathrm{Q}\rangle\rangle) = \langle \phi_\Gamma(\mathrm{Q}); \langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$ of the stack-based semantics. Let $R'' = \{r_{hd(\phi_Q(\mathrm{Q}))}\}$. Then the $(\rightarrow)$-SWITCH-rule is applicable iff $\mathcal{P}(R) < \mathcal{P}(R'')$. We first show that $R' = R''$. Let $\tilde{q}$ be the task within which u occurs. Then the condition $\mathcal{P}(R) < \mathcal{P}(R')$ implies that also $\mathcal{P}(r_{\tilde{q}}) < \mathcal{P}(R')$. Thus due to the PCP, $\langle u', R'\rangle = \langle q_e, \{r_q\}\rangle$ for some task $q \in \mathsf{Task}$ since a partially executed task of priority $\mathcal{P}(R')$ could not have been pre-empted by $\tilde{q}$. Since ϕ_Q preserves the relative ordering of tasks, and $\langle q_e, \{r_q\}\rangle = hd(\mathrm{Q})$, we conclude that also $q = hd(\phi_Q(Q))$.

It remains to prove that $\Phi\langle u', \langle R', \mathrm{Q}'\rangle\rangle$ equals the result of applying the $(\leftarrow)$-SWITCH-rule of the stack-based semantics to $\langle \phi_\Gamma(\mathrm{Q}); \langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$. The latter results in $\langle \phi_\Gamma(\mathrm{Q}); \langle v, R\rangle; \langle q_e, \{r_q\}\rangle, tl(\phi_Q(\mathrm{Q}))\rangle$. We have:

$$\begin{aligned}
\Phi(\langle v, \langle R', \mathrm{Q}'\rangle\rangle) &= \langle \phi_\Gamma(\mathrm{Q}'); \langle u', R'\rangle, \phi_Q(\mathrm{Q}')\rangle \\
&= \langle \phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})); \langle u', R'\rangle, \phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))\rangle \\
&= \langle \phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})); \langle q_e, \{r_q\}\rangle, \phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))\rangle
\end{aligned}$$

Regarding the stack, we verify that:

$$\phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})) = \phi_\Gamma(\mathrm{Q})\ ; \langle v, R\rangle$$

By the same arguments used for the head, all tasks in Q of higher priority than $\mathcal{P}(R)$ are of the form $\langle q''_e, \{r_{q''}\}\rangle$. Regarding ϕ_Γ, this implies that not only $\phi_\Gamma(\mathrm{Q}) = \phi_\Gamma(tl(\mathrm{Q}))$ but also $\phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q}))) = \phi_\Gamma(tl(\mathrm{Q}))\ ; \langle v, R\rangle$ – which together yield the desired equality.

Finally we must show that the queues match, i.e.,

$$\phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))) = tl(\phi_Q(\mathrm{Q}))$$

Since $\langle v, R\rangle$ is a partially executed task, $\phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))) = \phi_Q(tl(\mathrm{Q}))$ and $\phi_Q(\mathrm{Q}) = push(q, \phi_Q(tl(\mathrm{Q}))$. By preservation of the relative orderings of tasks, we know that q is the head of the translated queue thus $tl(\phi_Q(\mathrm{Q}) = \phi_Q(tl(\mathrm{Q})$. This completes the proof for the SWITCH-rule.

The proofs for the remaining corresponding rules follow by similar arguments. Here, we only show how the two disjoint cases of the KILL-rule are simulated by means of the NEXT- and RESUME-rule, respectively. When applying the KILL-rule, either the node u' at the head of the queue is an entry node of a task or not. If $u' = q_e$ for some $q \in \mathsf{Task}$, then its set of held resources equals $\{r_q\}$. The latter implies that $q = hd(\phi_Q(\mathrm{Q}))$, and, therefor, that the priority of r_q is higher than the priority of the set of held resources of all partially executed tasks in Q. This ensures that the priority condition of the NEXT-rule is satisfied while at the same time the priority condition of the RESUME-rule is falsified. If the head of Q is a partially executed task, it becomes the top of $\phi_\Gamma(\mathrm{Q})$, and the priority of its set of held resources is higher than (or equal and older) the static priority of all tasks in Q, in particular all fresh tasks. This ensures that the priority condition of the RESUME-rule is satisfied while at the same time the priority condition of the NEXT-rule is falsified. □

The stack-based semantics can naturally be extended with potentially recursive functions. The stack then has layers corresponding to pre-empted tasks followed by called functions which have not yet returned at the moment of pre-emption. Theorem 1 however also holds for queue-based and stack-based semantics when extended with function calls since they do not interfere with the PCP. In this case, the type of the translation Φ is given by:

$$\Phi : \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Stack}(N \times 2^{\mathsf{Res}})) \to \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Task})$$

Since there are finitely many tasks with finite activation numbers, the set of task queues possibly occurring in the stack-based semantics is finite. Similarly, the occurring sets of held resources are finite. Hence, the rules of the stack-based semantics can be interpreted as the transfer function of a *push-down* system where the sets of push-down symbols and states are given by $N \times 2^{\mathsf{Res}}$ and $\mathsf{Queue}(\mathsf{Task})$, respectively.

This also remains true when finite abstractions of local and global data are added. Therefore, techniques for analysing push-down systems [3] can be applied. In particular, reachability is decidable. When dealing with more expressive abstractions for deriving program invariants such as, e.g., affine equalities of variables, alias analysis, etc., techniques from interprocedural analysis [4] can be applied.

4 Related Work

The context- and synchronization-sensitive analysis of concurrent programs is generally undecidable even for simple analyses [13]. Therefore, one either over-approximates the interaction between threads, as in thread-modular model-

checking [6] and nested fix-point abstract interpretation [15]; or one places restrictions on concurrency, analysing precisely up to a fixed number of context switches [1, 12] or analysing structured parallelism only [5]. We obtain a precise analysis by exploiting properties of a specific framework for interrupt-driven concurrency. There exists relatively few works considering interrupt driven programs with priorities and to our knowledge none at all using the OSEK specification.

Closest to our work is the model used by Atig et al. [2]. They show that the control point reachability problem is decidable for asynchronous programs with pre-emption consisting of tasks with priorities. Thanks to the OSEK scheduling policy and the PCP we are able to keep the task pool structured as a queue and reduce the system to push-down automata, while they use a multi-set and reduce to petri-nets. While not mentioned by the authors, we believe that interrupts can be added to their model without needing to adjust the reduction.

In a number of papers, Kahlon et al. [7–9] discus model checking of push-down systems communicating via locks and asynchronous rendezvous (wait/notify). Using acquisition histories, they show that in the case of only nested locks the problem is decidable. While their model does not use priorities, this approach might also help to improve analysis of OSEK programs since nested use of locks is a desired, but not enforced, property of OSEK programs. Lammich and Müller-Olm [10] use a restricted semantics to generate monitor-consistent inter-leavings in order to find conflicting states in a system with dynamic thread creation and re-entrant monitors. We build in this idea by including priority conditions encoding the OSEK scheduling in our semantics.

Finally, Regehr and Cooprider, [14], present a source-transformation technique to turn interrupt driven embedded code into thread-based code. Applying normal race detection tools to the transformed code yields good results. They also introduce artificial interrupt locks to make interrupt disabling/(re-)enabling visible to the analysis. We slightly modify this idea to work with the PCP.

References

1. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS'09*, LNCS, vol. 5505, pages 107–123. Springer, 2009.
2. M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS'08*. IBFI, 2008.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, LNCS, vol. 1243, pages 135–150. Springer, 1997.
4. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.
5. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL'00*, pages 1–11. ACM Press, 2000.
6. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3):153–183, 2005.

7. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL'07*, pages 303–314. ACM Press, 2007.
8. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV'05*, LNCS, vol. 3576, pages 505–518. springer, 2005.
9. V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07*, LNCS, vol. 4590, pages 226–239. Springer, 2007.
10. P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS'08*, pages 205–220. springer, 2008.
11. OSEK/VDX Group. *OSEK/VDX Operating System Specification, Version 2.2.3*, 2005.
12. S. Qadeer and J. Rehof. Context-Bounded model checking of concurrent software. In *TACAS'05*, LNCS, vol. 3440, pages 93–107. Springer, 2005.
13. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.
14. J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, 2007.
15. H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.