Model Checking Trampoline OS: A Case Study on Safety Analysis for Automotive Software†

Yunja Choi∗

School of Computer Science and Engineering, Kyungpook National University, Korea

SUMMARY
This case study investigates methods and a process for software safety analysis of the Trampoline operating system using model checking. Model checking is an effective technique used to identify subtle problems in software safety using a comprehensive search algorithm. However, this comprehensiveness requires a large number of resources and is often too expensive to be applied in practice. This work strives to find a practical solution to model-checking automotive operating systems for the purpose of safety analysis, with minimum requirements and a systematic engineering approach for applying the technique in practice.

We present methods for converting the Trampoline kernel code into formal models for the model checker SPIN, a series of experiments using an incremental verification approach, and the use of embedded C constructs for performance improvement. The conversion methods include functional modularization and treatment for hardware-dependent code, such as memory access for context switching. The incremental verification approach aims at increasing the level of confidence in the verification even when comprehensiveness cannot be provided due to the limitations of the hardware resource. We also report on potential safety issues found in the Trampoline operating system during the experiments and present experimental evidence of the performance improvement using the embedded C constructs in SPIN. Copyright © 0000 John Wiley & Sons, Ltd.

KEY WORDS: Model Checking; Trampoline Operating System; Safety Analysis; Osek/Vdx; SPIN

1. INTRODUCTION

The operating system is the core part of automotive control software; its malfunction can cause critical errors in the automotive system, which in turn may result in loss of lives and assets. Much effort has been spent on developing a standard domain-specific development framework in automotive software [7, 34] to support a systematic and cost-effective safety analysis/assurance method.

So far, safety analysis for such systems is typically applied at the system level [33, 35] or at the small-scale source code level [16, 21, 36], separately with different kinds of focuses. Though international standards for the safe development of electronic/electrical devices, such as IEC 61508 and ISO 26262, recommend formal verification methods as a safety verification technique, practical experiences with processes or methods for applying formal methods in this domain are still rare, with little related literature on this matter [18, 20, 19, 32]. In fact, most existing work is focused on a certain aspect of an operating system, such as the scheduling algorithm and timing analysis, and requires extensive human expertise for effective verification, which is application dependent.

†This is an extended version of [11]
This work studies how automated formal verification techniques, such as model checking, can be systematically and efficiently used for the safety analysis of an automotive operating system under two basic premises: (1) The analyst does not necessarily have knowledge about the details of implementation nor extensive knowledge regarding a specific verification tool, and (2) general safety properties for automotive software are the target of the verification. With these premises, theorem proving or aggressive abstraction techniques for model checking, which require extensive domain knowledge during the verification process, would not be applicable. The aim of this work is to assess the applicability of model checking in a practical setting using only a systematic engineering approach. The study is conducted on the Trampoline [2] operating system, which is an open source operating system written in C and which is based on the OSEK/VDX [1] international standard for automotive real-time operating systems.

Since the safety of an operating system cannot be addressed without considering how it is used at the application level as well as at the system level, we take the system-level safety requirements and the functional requirements/constraints specified in the OSEK/VDX standard into account when we build the verification model from the kernel code: (1) the safety properties are identified from the automotive system level and then elaborated in terms of the Trampoline kernel code, and (2) the functional requirements and constraints in the OSEK/VDX standard are imposed on the task model, which is a unit of an application program.

The kernel code itself is faithfully converted into a formal model in PROMELA, the modeling language of SPIN [25], using existing C-to-PROMELA conversion methods [28]. However, methods for functional modularization, for task modeling to simulate the arbitrary behavior of a task, and for the context-switching mechanism are newly introduced. The approach is property-based because the model conversion process selectively extracts only those functions that have dependency relationships with respect to the safety properties. For this, we have developed a property-based code extractor on top of the static analysis tool Understand [3].

Five representative safety properties are verified or refuted using the model checker SPIN in three-step approaches. The initial light-weight verification focuses on error-finding using a generic task model with arbitrary behavior. After identifying potential safety issues and analyzing false negatives, the task model is further constrained to exclude illegal behaviors against the standard. The second step incrementally increases the complexity of the task model and analyzes the scalability of model checking. Further abstraction is performed in the third step using the embedded C constructs in PROMELA to reduce verification cost and improve model checking performance.

This approach enables us to identify a potential safety gap in the kernel code as well as several subtle issues that are difficult to identify using typical dynamic testing or theorem proving. The identified safety problem is confirmed by run-time testing using the counterexample sequence generated from the SPIN model checker as a test scenario. As expected, the model checking cannot scale with an indefinite number of system calls. Nevertheless, we anticipate that the typical complexity of a task w.r.t. its number of system calls is rather small in automotive ECUs, and, thus, the comprehensive verification result of a fixed number (15 in this case) of arbitrary API calls per task would be sufficient for providing confidence in the system.

The major contribution of this work can be summarized as follows:

1. A systematic model construction method together with a verification process is provided for model checking automotive operating system kernels on single-processor ECUs.
2. Safety analysis using model checking is demonstrated on the Trampoline operating system. This is the first extensive case study on safety analysis in general using model checking for an automotive operating system.
3. An incremental verification process and the new use of embedded C constructs in PROMELA are suggested to achieve better performance in practice.

The remainder of this paper is organized as follows: Section 2 provides some background knowledge for this work, including the safety properties in question, major requirements specified in the OSEK/VDX standards, and the model checking process for the Trampoline.
operating system. Section 3 introduces the conversion and modeling method for constructing a formal specification from the Trampoline kernel code. The result of the first-step verification is presented in Section 4, followed by the result of incremental verification in Section 5. Section 6 suggests an approach and experimental result for further improving model checking performance. We conclude with a summary and future work in Section 9, following a survey of related work (Section 7) and discussion of potential issues (Section 8).

2. SAFETY REQUIREMENTS OF OSEK/VDX

Since the Trampoline OS follows the OSEK/VDX standard, the safety requirements need to be analyzed under the requirements and constraints imposed by the standard.

2.1. OSEK/VDX requirements

OSEK/VDX [1] is an international standard specialized for automotive control systems. It is designed for stringent hardware resource constraints, removing unnecessary complexities or undesired behavior, since safety-critical systems such as automotive vehicles cannot afford such complexities. For example, it does not allow dynamic memory allocation, circular waiting for resources, and multi-processing. The following are some of the core requirements and constraints from the standard:

1. Dynamic memory allocation is not allowed; e.g., all tasks are required to allocate memory at system start-up time.
2. The maximum number of tasks per task type and the maximum number of activated tasks per task type are to be statically assigned and checked for each task activation.
3. The priority of a task must not change at run-time except when applying the priority ceiling protocol (PCP).
4. Only one task can be in execution at any given time, i.e., multi-processing is not allowed.
5. Task scheduling is based on a prioritized multi-level FIFO algorithm.
6. Resource de-allocation is based on an LIFO algorithm.

These restrictions seem quite severe for a generic operating system, but are reasonable for controlling an electronic control unit (ECU) of an automobile; typically, an automobile consists of up to a hundred of such ECUs, and the memory and energy requirements are thus quite stringent.

OSEK/VDX defines task models for user-defined tasks, which are the basic building blocks of an application program. A task interacts with the operating system through system calls. OSEK/VDX explicitly defines a total of 26 such APIs. Though it does not prevent users from modeling tasks with an unsafe sequence of system calls, it provides an error-checking mechanism and a list of error codes to prevent illegal usage of system calls. For example, if a task activates another task more often than the maximum activation limit of the task, the system call $\text{ActivateTask}$ is supposed to return an error code 4. In this way, illegal (unsafe) task design can be detected at runtime. This means that some of the safety issues may need to be rephrased from “Bad things never happen” to “If a bad thing happens, the corresponding error code shall be delivered”.

2.2. Safety requirements

Safety requirements for automotive operating systems are closely related to the safety of applications working on top of the operating system. Such safety requirements are analyzed in a top-down manner from the safety requirements of the overall automotive system, such as “An automobile control system shall change the direction of the wheels as intended by the driver”. Hardware faults, computational errors, errors in determining vehicle status, failures in protecting critical sections, and errors in task sequencing are some examples that may cause the vehicle control system to fail to change the direction of the wheels as intended by
the driver. Each such failure scenario is further analyzed using SFTA (Software Fault Tree Analysis) \cite{31} to identify software-level safety properties such as “Tasks and interrupt service routines shall not terminate while occupying resources” and “Tasks shall not loop indefinitely while occupying resources”.

Figure 1 shows an example of SFTA for automotive operating systems. Starting from a possible hazard, e.g., a delayed change of direction, the SFTA analyzes faults that may lead to the hazard. Each fault is again analyzed in a top-down manner to identify its possible cause. The left side of Figure 1 shows the first three levels of SFTA from the hazard “A. Delayed change of direction” and the right side of the figure shows the lower part of the SFTA from A.1. Safety properties are identified from the leaf component of the fault tree. For example, the property “Tasks shall not wait for events while occupying resources” is identified from component A.1.3.1.1. We have identified 56 safety properties in this way \cite{12}; five representative safety properties are given below:

- SR1. Tasks and ISRs shall not terminate while occupying resources.
- SR2. Tasks shall not wait for events while occupying resources.
- SR3. Tasks shall not wait for events indefinitely.
- SR4. Any activated tasks shall be executed in the end.
- SR5. A task with higher static priority always gets executed earlier than a task with lower static priority.

SR1 and SR2 are meant to eliminate the possibility of process deadlocks due to circular waiting of resources or indefinite waiting for a resource occupied by other tasks. SR1 is especially required since the operating system based on OSEK/VDX does not automatically reclaim allocated resources even after task termination. SR3 and SR4 are there to ensure the absence of starvation either by mismatched event sequences, mistakenly designed task priorities, or process deadlocks. SR5 is one of the safety properties identified from the SFTA A.1.3.2.2 to ensure that there is no dynamic lowering of the priority that may unexpectedly change intended execution order.

Unlike functional requirements, the verification of safety requirements mostly requires showing the absence of behavior rather than the existence of behavior, and therefore, typical scenario-based testing approaches are not sufficient for this purpose. That is why most existing approaches, including safety standards, recommend formal verification as an alternative and necessary solution.

2.3. Model checking Trampoline

Model checking \cite{14} is an automated formal verification technique based on an exhaustive search of the system state-space. It requires the target system to be formally modeled and the properties in question to be formalized in logical terms. This work uses the SPIN model checker \cite{25}, since it is one of the most frequently used model checkers with extensive experience and support from which we take useful hints for this case study \cite{20, 28, 39}.
Figure 2. Model checking process

Figure 2 illustrates the overall verification process using model checking: the initial step consists of two manual activities, construction of the model from the Trampoline kernel code and identification of the safety properties from the software fault tree analysis. Since the verification of the Trampoline operating system requires formal modeling of the kernel code itself as well as modeling of the user task, the PROMELA model includes a system model constructed from the Trampoline kernel and a generic task model for user tasks constrained by the OSEK/VDX standard.

The PROMELA model is validated using the SPIN simulator via random simulation and is then verified with respect to each safety property using the SPIN model checker. In this step, light-weight model checking is applied using limited resources, e.g., 4 Gbytes of system memory. The process ends if all the properties are verified in this step.

Each refuted safety property is analyzed via the counterexample replay facility provided by the simulator. If it turns out that the counterexample is a false alarm, due to mistakes in the model, for example, then the correction is made to the model directly and re-verification is performed. Otherwise, we generate a test scenario from the counterexample and perform runtime testing of the Trampoline code on Linux to confirm that the counterexample does, in fact, reflect an actual property violation. For those properties that are neither verified nor refuted, due to the limitation of the resource, the process goes to the second step. In this step, the verification is performed iteratively using larger resources by constraining verification conditions and relaxing them incrementally. The incremental verification process includes the application of embedded C constructs in PROMELA to improve scalability as the third step of the verification process.

The first step of the verification was performed on a PC with Pentium III CPU and 4 Gbytes of memory for the quick and easy identification of problems and false alarms. After addressing issues identified from the initial verification, more extensive verification was performed on a SUN workstation with 30 Gbytes of memory. SPIN version 6.0.1 and its graphical runtime environment iSpin 1.0.0 were used for the verification.

3. CONVERSION AND MODELING

A formal model of the Trampoline OS was constructed using one-to-one translation from C constructs to PROMELA constructs using existing approaches [24, 29]. This section explains the translation and modeling approaches unique to our work, while omitting details of conventional translation rules.
3.1. Overall approach

The Trampoline operating system kernel consists of 174 functions, which comprise a total of 4,530 lines of code. However, many of the functions act as wrappers, abstracting details on which implementation function is called inside, or as tracing code for debugging. This also includes platform-specific functions for hardware access and functions for emulating environments. Since not all of them are related to the core service of the operating system, we first identify the core functions contributing to the safety properties in question with the following guidelines:

1. Eliminate wrapper functions and convert only the actual code.
2. Eliminate tracing functions for debugging.
3. Abstract hardware-dependent code.
4. Eliminate emulating functions.

We first identify core functions that have a dependency relationship with major operating system APIs. The core functions are classified by their service types and are grouped into modules according to the classification. Each module and its member functions are converted into corresponding PROMELA constructs.

3.2. Extracting core functions and modularization

We have identified 8 out of 26 APIs as being closely relevant to the safety properties in question – ActivateTask, TerminateTask, ChainTask, Schedule, GetResource, ReleaseResource, SetEvent, WaitEvent. The identification of APIs related to the safety properties is based on the OSEK/VDX specification explaining the functionality of the APIs. For the most general cases, we assume that all the interrupts are enabled and the handling of alarms is static.

A call graph is generated for each identified API using the Understand code analyzer [3], through which its dependent core functions are identified manually and extracted automatically. A code extractor is implemented on top of Understand for this purpose.

Figure 3 (a) is an example of a functional call graph identified from a core API, ActivateTask, where the function ActivateTask is defined as returning the result of tpl_activate_task_service. The identified core functions from the call graph are highlighted with a dark box. For example, the function tpl_call_error_hook is a wrapper function that simply calls the ErrorHook function after checking that the function is not called recursively. The function tpl_switch_context is a hardware-dependent function accessing physical memory locations. As explained in a later section, we abstract the hardware-dependent context switch mechanism at the software level by introducing explicit switching points in the task model.

Figure 3. Modularizing Trampoline functions
Table I. Categorizing eliminated functions

<table>
<thead>
<tr>
<th>Category</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapper</td>
<td>26</td>
</tr>
<tr>
<td>Hardware-dependent</td>
<td>16</td>
</tr>
<tr>
<td>For tracing and debugging</td>
<td>26</td>
</tr>
<tr>
<td>For interfacing with emulator</td>
<td>6</td>
</tr>
<tr>
<td>For error hooks</td>
<td>11</td>
</tr>
<tr>
<td>Core functions</td>
<td>89</td>
</tr>
<tr>
<td>Functions independent of the 8 APIs</td>
<td>49</td>
</tr>
<tr>
<td>Extracted</td>
<td>40</td>
</tr>
</tbody>
</table>

Table II. A mapping for conversion

<table>
<thead>
<tr>
<th>Kernel constructs</th>
<th>Promela constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A module</td>
<td>Proctype</td>
</tr>
<tr>
<td>A member function of a module</td>
<td>A composite state under a label</td>
</tr>
<tr>
<td>A common function</td>
<td>An inline function</td>
</tr>
</tbody>
</table>

Table I summarizes the number of functions eliminated from the kernel modeling. As a result, a total of 40 functions are modeled out of 174 functions.

The core functions are collected and classified into six categories: task service, resource management, event management, scheduler, process service, and common service functions. Figure 3 (b) illustrates the modular structure based on the categorization. For example, `tpl_activate_task` identified from the `ActivateTask` API becomes a member function of `task_service`. `tpl_init_proc`, `tpl_put_new_proc`, `tpl_get_proc`, and `tpl_put_preempted_proc` are members of `proc_service`. On the other hand, `tpl_get_internal_resource` belongs to the module for common functions. In the figure, the `task` is not a part of the operating system kernel, but acts as a user of the operating system. We have two types of tasks, `basic` and `extended`, as defined in OSEK/VDX. Information on individual tasks, such as task id, priority, resources, etc., is maintained in separate data structures as will be explained in Section 3.4. The module for common functions, which is used by all the modules, is not included in the figure.

3.3. Conversion of modules

There are three possible ways of modeling a C function in PROMELA; it can be modeled (1) as a synchronized communicating process, `proctype`, with message passing, (2) as an inline function, and (3) as an embedded C code that is external to the model. The third case does not contribute to the model’s statespace during the model checking process, but is executed separately to provide only the function values to the model. Though the third method may be the most effective one in reducing verification resources, it must be used with care since it may merge several transitions into one, possibly losing important behavior accidently. Therefore, we take a mixture of the first and the second methods, while the third approach is selectively applied later in the third verification step to improve performance.

Each module is converted into a synchronized communicating process, `proctype`, where its member functions are translated into a collection of labeled transition systems with a common initial state. The common service functions are converted into inline functions for reasons of simplicity. Table II summarizes the mapping between modules and PROMELA constructs.

For example, the following is a fragment of the Promela model converted from the module `scheduler`.

```plaintext
1: proctype scheduler(){
2: // declaration of local variables
   ...
```
The module scheduler is modeled as a communicating process, which is defined as `proctype` in PROMELA and is initially in the `get_message` state. As a function call is received through a message channel `mch`, it jumps to the corresponding label, performs the corresponding functions, returns a message to the caller, and jumps back to the initial state. For example, if the `start` message is received (line 5), it jumps to the `start_schedule` label (line 11), where it starts performing the function `tpl_start_schedule`. After finishing the corresponding function, it returns a return message to the caller (line 15) and jumps back to its initial state (line 16). Each `proctype` can be instantiated as an independent process; these processes are synchronized with each other through message passing.

Functions often used by other functions are categorized as common functions and modeled using the inline construct in PROMELA. The following is an example for the function `tpl_init_proc`, which is used in the function `tpl_start_schedule` (line 13).

```
inline tpl_init_proc(proc_id){
    // assign static values to the dynamic process table
    tpl_dyn_proc_table[proc_id].ppriority =
        tpl_stat_proc_table[proc_id].base_priority;
    ...
    // initialize process context by calling another inline function
    tpl_init_context(proc_id);
}
```

In this way, the user-level interactions remain traceable through message passing sequences, while verification costs are reduced by minimizing the number of communicating processes. The PROMELA model translated from the core kernel code includes a minimum of 6 communicating processes and 18 inline functions comprising a total of 1,500 lines of code. The number of communicating processes may increase depending on the number of task types.

### 3.4. Conversion of data structures and variables

The Trampoline kernel maintains (1) information on the currently running and the previously executed task objects, including both static and dynamic descriptions of tasks, and (2) a service call description that records service id, resource id, task id, event masks, and so on.
As illustrated in Figure 4, the kernel receives a service call from the task in the process whose static/dynamic task information is recorded in the *kernel process information block*. The kernel process information block is used to refer to the static/dynamic description of the process and is used to perform rescheduling, put the current task into the ready queue, or get and start a task from the ready queue. It may also be used to change the priority of the currently running task if the service call is to allocate a resource whose ceiling priority is greater than the task’s current priority.

Those data structures are faithfully converted into PROMELA global data types after replacing pointers with array indices and pointer assignments with the deep copy of the corresponding arrays. Primitive variable types are converted into the corresponding PROMELA basic types such as bit, byte, int, and unsigned. For example in Table III, the data type `tpl_priority_level` in Trampoline is converted into PROMELA data types by (1) changing the pointer type into a fixed-sized array and (2) replacing the `u8` type (unsigned character) with the `byte` type. Since the pointer type is converted into a fixed-size array, an assignment of `TPL_PRIORITY_LEVEL` type variables requires copying all the values of the array elements. We note that the data type for `fifo` is changed from `tpl_proc_id`, which is a signed character, to `TASK_INFO`, which is a `struct` type composed of a process id and a switching point. This is an exceptional case; we added the field for storing the context switching point to support simulation of context switching. This will be explained further in Section 3.6 and Section 3.7.
We did not apply aggressive abstractions on the data structure and global variables for two reasons: First, we did not have a complete understanding of the implementation details. Aggressive abstraction is effective in reducing verification costs, but can be dangerous if performed by someone who does not understand the rationale behind the implementation details. Second, aggressive abstraction makes the counterexample analysis difficult due to the large difference between the verification model and the actual code.

3.5. A task model for comprehensive scenario generation

As an operating system normally stays idle till a user request is received, behavioral problems in an operating system can only be identified through user tasks. Therefore, comprehensive modeling of a user task is also important for the verification of an operating system. We achieve the comprehensive modeling of a generic user task by simulating arbitrary deterministic task behavior with non-deterministic behavior. In our approach, the generic task generates all possible interaction scenarios with the Trampoline kernel code in the verification process, which is constrained by the OSEK/OS specifications, but not constrained by a specific user task.

OSEK/VDX specifies that a task may have four states: suspended, ready, running, and waiting. A notable fact is that OSEK/VDX requires all tasks to be designed statically and to be loaded in the memory as the operating system starts up. Therefore, all tasks are in suspended states initially. We have elaborated the original task model by refining the running state with details of possible interactions with the Trampoline kernel code.

Figure 5 shows the refined task model for the extended task type; each sub-state in the running state is labeled with a possible system call that can be performed in the state. Basically, all the sub-states in the running state are strongly connected with transitions except for some restrictions imposed by the basic requirements of the OSEK/VDX standard. Examples of such requirements are as follows:

1. A task must not terminate or chain another task while holding resources, i.e., a task cannot transit to TerminateTask or ChainTask from the getResource state.
2. A task must not be in a waiting state while holding resources, i.e., a task cannot transit to WaitEvent from the getResource state.
3. A terminated or chained task goes into the suspended state.

Figure 5 reflects those constraints in the task transitions; e.g., after the getResource API call, a task transits to sp4, from which a non-deterministic choice to transit to ReleaseResource, ActivateTask, or SetEvent can be made, but it cannot transit to TerminateTask, ChainTask, or WaitEvent from sp4. Self-transitions are allowed, though not explicitly depicted in the figure, for each state except for the WaitEvent, ChainTask, and TerminateTask states, which are drawn in dashed lines.

We note that the generic task model excludes only obvious illegal behaviors from a user task but allows potential design mistakes; for example, the model does not restrict the number of calls to ActivateTask and allows a task to terminate while holding a resource, such as sp0 → getResource → SetEvent → sp1 → TerminateTask. The purpose is to construct a generic task that subsumes potential corner cases in task design while reducing unnecessary complexities, rather than defining a perfect user task. In this way, we can identify potential issues of the kernel code that may be caused by illegal usage patterns or bad task design.

We also note that the non-deterministic choice is not only available for the system calls but also for the values of the parameters for each system call; this may include infeasible selection of parameter values depending on the context, which matches with our purpose to find corner cases.

The task model for the basic task type is the same as the one for the extended task type, except that the WaitEvent state and its related transitions are removed.
3.6. Software simulation of context switch

Since Trampoline supports several hardware platforms, it is expensive to model each platform for the safety analysis of the software part. We abstract the hardware-dependent code to reduce verification costs and to find platform-independent software problems.

For example, the following is the Trampoline code for a context switch in the POSIX environment:

```c
void tpl_switch_context(const tpl_context * old_context,
                        const tpl_context * new_context){
    if(NULL==old_context){
        _longjmp((*new_context)->current, 1);
    }
    else if(0==_setjmp((*old_context)->current)){
        _longjmp((*new_context)->current, 1);
    }
    return;
}
```

The code stores the current context and retrieves the new context from the physical memory whenever a task is preempted. We simplified this mechanism by introducing explicit switching points in the task model and by annotating each task in the kernel process information block with its switching point.

Figure 5 illustrates the five explicit switching points in the task model, $sp_0$, $sp_1$, $sp_2$, $sp_3$, and $sp_4$ in the running state. Since each system call is protected by an internal locking mechanism, we safely assume that context switching occurs only in-between system calls. When context switching occurs, the switching point is stored in the kernel process information block for the switched task (Table III). The switched task is rescheduled according to its priority, and resumes from the switching point it has left; the history mark $H$ in the running state means that it remembers the last active sub-state.

The following is a sample PROMELA code for the context-switching process: First, it checks whether the newly activated task has higher priority than the currently running task. If so, it puts the task into the ready queue and copies its information, including the switching point, into the kernel process information block.

```promela
if ::tpl_h_prio > tpl_kern.running.ppriority ->
    mch!put,schedule_from_running;
    param!tpl_kern.running_id,0;
    COPY(tpl_kern.old, tpl_kern.running);
    COPY(tpl_kern.s_old, tpl_kern.s_running);
```

Figure 5. A task model with explicit switching points (Extended Task)
The next code shows the way context switching is modeled in Promela: If the kernel determines that switching is required, it first sends the switch command to the currently running task through the channel cch, then sends the start command to the highest-priority task in the ready queue, and finally sends the switching point where it can resume execution.

```promela
if ::tpl_kern.need_switch != NO_NEED_SWITCH ->
  id = tpl_kern.s_old.context;
  cch!switch, id;
  id = tpl_kern.s_running.context;
  cch!start, id;
  cch!startAt, tpl_kern.s_running.switchPoint;
:: else ->skip;
fi;
```

Typically, it is not necessary to explicitly model the context switching behavior because processes in Promela are already designed to arbitrarily interleave with each other, simulating arbitrary context switching at the statement level. However, arbitrary interleaving includes invalid behaviors, such as context switching in the middle of executing system calls. We remove such invalid behaviors by explicitly modeling context switching. This also enables us to simulate multiple tasks using one generic task model as explained in the next section.

### 3.7. Simulation model for multiple task activation

A task is modeled in proctype in Promela, which can be activated multiple times. Multiple activations of a proctype typically result in the creation of multiple independent copies of a process with the same behavior. However, as OSEK does not allow parallel execution of multiple copies of a task, we allow only one active proctype per task type and simulate multiple tasks/activations by tracing the execution point of each activated copy.

Consider the three active processes of the same task type in Figure 6 (a) when multiple activations of proctype are allowed. They have exactly the same behavior but are executed independent of each other. However, as OSEK does not allow multi-processing, only one of them can execute at any given time, and, thus, these multiple copies can be serialized by keeping track of the execution points for each of them. For example, the running sequence $T_2: A \rightarrow T_2: B \rightarrow T_1: A \rightarrow T_1: B \rightarrow T_3: C \rightarrow T_3: D \rightarrow T_2: B \rightarrow \ldots$ can be simulated by one active proctype by recording the context switching point in the ready queue together with the task id as shown in Figure 6 (b).

1. $T_2$ is activated and starts execution.
2. $T_1$ and $T_3$ are activated and (task_id, switching_point) information for each of them is stored in the ready queue. By default, the switching point is the initial state.
3. $T_2$ goes to the ready/waiting state at $B$ and $T_1$ starts execution from switching point $A$.

Now the information in the ready queue is $\{(T_{3}, A), (T_{2}, B)\}$. 
4. \( T_1 \) goes to the ready/waiting state at \( C \) and \( T_3 \) starts execution from switching point \( A \).
Now the information in the ready queue is \( \{(T_2, B), (T_1, C)\} \).

5. \( T_3 \) goes to the ready/waiting state at \( D \) and \( T_2 \) starts execution from switching point \( B \).
Now the information in the ready queue is \( \{(T_1, C), (T_3, D)\} \).

6. ...

Figure 6 (b) is an illustration of such a simulation. We modified the ready queue in the PROMELA model to maintain the information on the switching point for each activation of a task so that one active procotype simulates multiple processes of the same task type. Considering that the number of active processes in PROMELA is a critical factor in verification cost, this modification, which is intended to explicitly handle the switching points, is a minor sacrifice to improve scalability.

4. INITIAL VERIFICATION RESULT

Table IV is a list of the safety properties introduced in Section 2 and their corresponding formal specification together with the verification results. The formal specification is written in temporal logic LTL, which is a propositional logic with temporal operators. Given two propositional formulas \( x \) and \( y \), the meaning of each temporal logic operator used in the safety property is as follows:

- \( \square x \): \( x \) is always true for all execution paths
- \( <> x \): \( x \) is true at some time in future states
- \( xUy \): \( x \) is true until \( y \) is true

For example, the formal specification for \( SR_1 \) can be interpreted as "For all execution paths, if TerminateTask or ChainTask is called and the calling task has a resource, then an error will be set at some time in future states". We note that the LTL formulas are somewhat simplified from the original version; e.g., a variable \( i \) is used in the LTL formula of \( SR_3 \) and \( SR_4 \) to specify them generally, but an explicit value is used for actual verification, such as \( \square((\text{wait\_id} == 0) \rightarrow <> (\text{tpl\_kern\_running\_id} == 0)) \), for each task identifier \( i \). Also, task\_has\_Resource of \( SR_2 \) and \( \text{wait\_id} \) of \( SR_3 \) are macros representing corresponding values from the Trampoline data structure. \( SR_5 \) is specified as \( SR_5_1 \) in the initial formal specification, but is strengthened to \( SR_5_2 \) after a counterexample analysis is performed; \( SR_5_1 \) specifies that it is always the case that if \( \text{task}_i \) has lower static priority than \( \text{task}_j \), and if \( \text{task}_i \) and \( \text{task}_j \) are in the ready state at the same time, then \( \text{task}_i \) does not get into the running state before \( \text{task}_j \) unless an error occurs.

We performed initial verification using the kernel model and the generic task model. The result shows that \( SR_1, SR_2, \) and \( SR_5_2 \) were neither verified nor refuted because they ran out of memory. \( SR_3, SR_4, \) and \( SR_5_1 \) were refuted. The following is a detailed analysis of the result.

4.1. A potential safety issue due to under-guarded APIs

The safety property \( SR_3 \) was refuted in 0.16 seconds, after searching 29,116 states and 34,979 transitions, consuming 9.3 Mbytes of memory. The counterexample task scenario identified by SPIN against the property is as follows:

1. Task \( t_1 \), which is an autostart task with priority 1, starts at system start-up time. It allocates resource 1 and activates task \( t_2 \), releases resource 1, and terminates afterwards.
2. Task \( t_2 \) has priority 5. Therefore, it preempts \( t_3 \) as soon as it is activated by \( t_1 \). \( t_2 \) activates task \( t_3 \), waits for event 2, and terminates afterwards.
3. Task \( t_3 \) has priority 2. Once it is started, it activates task \( t_4 \), sets event 0 for \( t_4 \), and terminates afterwards.
<table>
<thead>
<tr>
<th></th>
<th>Formal specification</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR1</td>
<td>```((TerminateTask</td>
<td></td>
</tr>
<tr>
<td>SR2</td>
<td><code>(WaitEvent &amp; taskhasResource -&gt; &lt;&gt; (error))</code></td>
<td>incomplete</td>
</tr>
<tr>
<td>SR3</td>
<td><code>((wait_id == i -&gt; &lt;&gt; (tpl_kern.running_id == i))</code></td>
<td>fail</td>
</tr>
<tr>
<td>SR4</td>
<td><code>((dy_n_proc_table[i].state == ready -&gt; &lt;&gt; (tpl_kern.running_id == i))</code></td>
<td>fail</td>
</tr>
<tr>
<td>SR5</td>
<td><code>((stat_proc_table[i].priority &lt; stat_proc_table[j].priority &amp; dy_n_proc_table[i].state == ready &amp; dy_n_proc_table[j].state == ready)</code></td>
<td>fail</td>
</tr>
<tr>
<td></td>
<td>```-&gt; dy_n_proc_table[i].state != running U (dy_n_proc_table[j].state == running</td>
<td></td>
</tr>
<tr>
<td></td>
<td>```-&gt; dy_n_proc_table[i].state != running U (dy_n_proc_table[j].state == running</td>
<td></td>
</tr>
</tbody>
</table>

4. Task $t_4$ has priority 4. Once it is started, it waits for event 0, sets event 2 for $t_2$, and terminates afterwards.

The scenario looks normal and it is expected that all the tasks terminate normally. As Figure 7 illustrates, however, the SPIN verifier finds an abnormal behavior for this task scenario: $t_4$ waits for event 0 indefinitely even though task $t_3$ sets event 0 for $t_4$, and, thus, $t_4$ cannot run to set event 2 for task $t_2$, which again makes $t_4$ wait for the event indefinitely. As a result, two of the four tasks cannot terminate normally. It turns out that the source of the problem is in the encoding and checking mechanism for events in the Trampoline kernel code, as shown in the following code fragment from the Trampoline kernel.

```c
1:tpl_status tpl_set_event(tpl_task_id task_id, tpl_event_mask in_event){
  2: ....
  3: if((events->evt_wait & in_event)!=0){
      4: ....
      5: // wake up and put the waiting process in the ready queue
      6: ....
      7: }
      8: ....
  9:}
```

All events are represented in an 8-bit mask; if a task calls the `WaitEvent` for event $i$, then the $i_{th}$ bit of the event mask is set. When a task sets the event $i$ for the waiting task, it calls `tpl_set_event` with the task identifier and the event number. As stated in line 3, it performs a bitwise-and operation of the event mask and the event number to check that a task is indeed waiting for the event. However, this encoding and checking mechanism only works correctly when the event number is greater than 0; `WaitEvent(0)` does not have any effect on the event mask since the bitwise-and operation of the event mask and the event number are always equal to 0. In this case, the lines between 4 and 6 are not executed, and, thus, cannot wake up the task waiting for event 0.

In fact, according to the OSEK/VDX specifications, the events are supposed to be declared with names, such as `evt1, evt2`, etc., and it is assumed that those names are used when calling the `WaitEvent` and `SetEvent` system services instead of using event numbers, such as `WaitEvent(evt1) and SetEvent(t2, evt1)`. Trampoline internally converts the event names into numbers starting from 1 so that the identified erroneous behavior cannot be possible. Nevertheless, it is allowed to use event numbers directly in the user task in Trampoline, i.e., we
can still code \textit{WaitEvent}(0) and \textit{SetEvent}(t2, 0), without getting warnings or error messages. We anticipate that this is a typical case of a safety gap; considering that a safety problem is mostly caused by unexpected corner cases, it is always recommended to safeguard potential issues. For example, this type of counterexample should be avoided if any one of the following conditions is satisfied:

1. Application programmers follow the assumed method of coding and design.
2. A compiler that checks for the correct use of the \textit{SetEvent} and the \textit{WaitEvent} APIs from the user task is provided.
3. Pre- and post-conditions for each kernel function are defined and checked.
4. Run-time error handling for invalid input is implemented for each API.

Since we cannot afford safety failures in automotive systems, at least one of the systematic guarding and checking mechanisms, other than human responsibility, is necessary.

4.2. Potential safety issues due to unsafe task scenario

The safety property \textit{SR4} is also refuted and the counterexample analysis revealed two types of potential design errors in user tasks: (1) over-activation of a task beyond the maximum activation limit, and (2) the existence of an infinitely running task with higher priority than the task waiting in the ready queue. A representative example for each case is given below.

- **Type 1**: Task \( t_1 \) activates task \( t_0 \) multiple times over the maximum activation limit when \( t_0 \) has lower priority than \( t_1 \). This causes multiple activation errors and the system stops without running \( t_0 \).
- **Type 2**: Let \( t_0, t_1, t_2, \) and \( t_3 \) be tasks and \( r \) a resource owned by \( t_2 \) such that \( \text{Prio}(t_0) < \text{Prio}(t_1) < \text{Prio}(t_2) < \text{Prio}(t_3) \), where \( \text{Prio}(t) \) is the priority of task \( t \). Suppose \( t_0 \) first runs and gets the resource \( r \), then activates \( t_1 \) and \( t_3 \), in that order. Since the OSEK/VDX standard uses the Priority Ceiling Protocol, the priority of \( t_0 \) becomes equal to that of \( t_2 \), which owns \( r \). Thus, \( t_1 \) stays in the ready queue while \( t_3 \) preempts \( t_0 \) and gets in the running state. If \( t_3 \) activates itself by calling \textit{ChainTask}(\( t_3 \)), then \( t_3 \) is executed infinitely many times while \( t_1 \) is starving.

The \textit{Type 1} case can be detected using the error handling mechanism, and thus, the counterexample can be considered as a false negative, as \textit{SR4} can be rephrased and verified as

\[
\text{SR4}_2: 
\left( \text{dyn_proc_table}[i].\text{state} = \text{ready} \rightarrow <> (\text{tpl.kern.running.id} = i \mid \text{error}) \right). 
\]

The \textit{Type 2} case is more problematic since the starvation may not be obvious at the task design level, but no error-handling mechanism is provided to detect such a situation since self-reactivation is not prohibited by the OSEK/VDX standard. We may be able to avoid the \textit{Type 2} case by prohibiting self-activation using \textit{ChainTask} or by performing static/dynamic infinite loop detection. Though it is possible for some applications that infinite reactivation of
a task is necessary, we anticipate that such a case is only exceptional and can be handled by an application-specific way, i.e., by using a deterministic task sequence designed for the specific case.

4.3. Unsatisfied property due to the use of the Priority Ceiling Protocol

The safety property $SR_5$, “A task with higher static priority always gets executed earlier than a task with lower static priority”, formally specified as $SR_5_1$, is not satisfied by the Trampoline OS because of the use of Osek’s Priority Ceiling Protocol (PCP) [1]. The Osek PCP is designed to avoid the problem of priority inversion and deadlocks by statically assigning a ceiling priority for each resource and temporarily raising the priority of a task while the task allocates a resource whose ceiling priority is higher than the priority of the task. The following is a counterexample scenario identified by SPIN:

1. Task $t_1$, which has static priority 1, runs first and activates task $t_2$, which has static priority 5.
2. $t_2$ preempts $t_1$ and waits for event $evt1$.
3. $t_1$ resumes and allocations resource $r$ whose ceiling priority is 6. Then, the priority of $t_1$ is temporarily promoted to 6.
4. $t_1$ activates task $t_3$, which has static priority 7, and is preempted. Now, $t_1$ is in the ready state.
5. $t_3$ sets event $evt1$ for $t_2$. Now, $t_2$ is in the ready state.
6. $t_3$ terminates. Then $t_1$ goes to the running state first, since its priority is temporarily higher than that of $t_2$.

The Trampoline OS contains the potential fault A.1.3.2.2 identified in SFTA (Figure 1), “dynamically lowering the priority in runtime”. Therefore, we cannot guarantee that there is no delay for a task with relatively higher static priority in general. As in the case of checking starvation in the event of an infinite task sequence, application-specific verification of this property can be used to validate a given task design, e.g., to ensure that a certain task maintains highest priority under any circumstances.

$SR_5_2$ is a stricter specification of $SR_5$, adding the precondition that the priority of task $j$ is higher than the maximum ceiling priority of all resources. The model checker did not find any counterexample for the property within the given resources. This will be discussed in more detail in the next section.

4.4. Incomplete verification

Three out of the six properties, $SR_1$, $SR_2$, and $SR_5_2$, could be neither verified nor refuted due to resource limitations. For example, the verification for $SR_1$ quickly ran out of memory on a PC with 4 Gbyte of memory. Even on the SUN workstation with 30 Gbytes of memory, SPIN reported out-of-memory; Table V shows the comparative performance data using different verification options, $-DCOLLAPSE$, $-DHC4$, and $-DBITSTATE$, for the verification of $SR_1$. The first two cases ran out of memory. The bitstate-hashing option trades completeness for less memory usage, but the time required for the verification was too high, over 7 days, whereas the resulting search coverage, indicated by the hash factor $577$, was not so high$^*$. Though SPIN did not report any safety errors during the search, we could not conclude that there is no safety error, since the search was incomplete mainly because it required more memory than available.

$^*$A hash factor between 10 and 100 has an expected coverage from 84% to 98%, and SPIN recommends trusting the result of a bitstate search only if the hash factor is greater than 100 [23].
## 5. INCREMENTAL VERIFICATION AND PERFORMANCE

Though the first-step verification was successful in finding potential safety issues in the kernel code as well as in the task design, we still need a better answer for those properties where verification was incomplete.

The inefficiency of the first-step verification results from two factors: (1) The Trampoline kernel itself is too large in the statespace, and (2) the task model used in the first-step verification is too generous in that it allows an arbitrary number of system calls in a task as well as infinite task sequences. Since we aim at avoiding aggressive abstractions on the model, we focus on the second factor to find room to improve performance and efficiency. Below are some observations made.

1. A task normally uses a limited number of system calls in practice.
2. Many counterexamples are caused by atypical infinite task sequencing.

Therefore, the second-step verification puts more constraints on the task model to exclude such cases and tries to get a meaningful measure for comprehensive verification. The following are the constraints imposed on the initial generic task model:

- **C1.** The number of system calls is limited per task, producing more conservative scenarios but still mimicking an arbitrary behavior of a task.
- **C2.** Over-activation of a task is prohibited.
- **C3.** Self-activation using `ChainTask`, i.e., calling `ChainTask(t)` from task `t`, is prohibited.

C1 is imposed on the task model by inserting a counter `cnt_APIcalls` for API calls, incrementing the counter for each API call, and imposing a guarding condition `cnt_APIcalls < CallLimit` for each transition. Note that even though the number of system calls is limited, non-determinism is still retained, and so is the arbitrary behavior of the task. C2 is imposed on the task model by inserting a guarding condition `active_count < max_activation_count` for the transitions going into the `ActivateTask` or `ChainTask` states so that those system calls are called only when the current activation count is below the threshold. C2 is not a necessary constraint since the over-activation is to be caught by the error-handling mechanism in the Trampoline OS, but it helps to reduce uninteresting internal transitions, and, thus, reduce verification costs. The problem with atypical infinite task sequencing is addressed by limiting the number of API calls and by imposing C3. We note that infinite task sequencing is still allowed by activating each other, but an infinite loop within a task and infinitely self-activating tasks are not allowed under these constraints.

With this more constrained task model, all three properties that were incomplete in the initial verification, $SR_1$, $SR_2$, and $SR_5$, are verified within the given resources. $SR_4$, which was refuted due to the infinite task sequencing, is also verified.

Table VI shows the performance of model checking the safety property $SR_2$ as the number of system calls increases from 3 to 15. The columns from left to right represent the number of API calls, the depth of the verification search, the number of states explored, the number of transitions, the amount of memory used in Megabytes, and the time required to finish verification in seconds. The $-DHC4$ option is used for the experiment. We note that around 29 Gbytes of memory are consumed for comprehensive verification, with a maximum of 15 system calls per task. That is, 15 system calls per task are the limit of comprehensive verification in the second verification step.

### Table V. SPIN verification options and performance

<table>
<thead>
<tr>
<th>Option</th>
<th>Depth</th>
<th>States</th>
<th>Transitions</th>
<th>Memory(Mbytes)</th>
<th>Time(seconds)</th>
<th>H/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DCOLLPASE</td>
<td>7,160,490</td>
<td>4.16e+08</td>
<td>5.78e+08</td>
<td>30,926.651</td>
<td>3.34e+04</td>
<td>N/A</td>
</tr>
<tr>
<td>-DHC4</td>
<td>9,494,460</td>
<td>6.64e+08</td>
<td>9.2e+08</td>
<td>30,959.950</td>
<td>1.93e+04</td>
<td>N/A</td>
</tr>
<tr>
<td>-DBITSTATE</td>
<td>9,999,999</td>
<td>1.3e+10</td>
<td>1.96e+10</td>
<td>9,725.277</td>
<td>6.43e+05</td>
<td>5.27</td>
</tr>
</tbody>
</table>

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using stvrauth.cls

DOI: 10.1002/stvr
Table VI. Performance of incremental verification

<table>
<thead>
<tr>
<th>APIs</th>
<th>Depth</th>
<th>States</th>
<th>Transitions</th>
<th>Memory(Mbytes)</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>185,359</td>
<td>8.00e+06</td>
<td>1.26e+07</td>
<td>840.100</td>
<td>253</td>
</tr>
<tr>
<td>5</td>
<td>1,041,301</td>
<td>4.20e+07</td>
<td>6.57e+07</td>
<td>2,371.397</td>
<td>1.28e+03</td>
</tr>
<tr>
<td>7</td>
<td>2,180,863</td>
<td>1.12e+08</td>
<td>1.75e+08</td>
<td>6,017.955</td>
<td>3.42e+03</td>
</tr>
<tr>
<td>9</td>
<td>3,396,568</td>
<td>1.92e+08</td>
<td>2.97e+08</td>
<td>9,059.460</td>
<td>6.23e+03</td>
</tr>
<tr>
<td>11</td>
<td>4,934,305</td>
<td>3.02e+08</td>
<td>4.67e+08</td>
<td>17,213.662</td>
<td>9.90e+03</td>
</tr>
<tr>
<td>13</td>
<td>6,794,956</td>
<td>4.40e+08</td>
<td>6.79e+08</td>
<td>22,467.666</td>
<td>1.39e+04</td>
</tr>
<tr>
<td>15</td>
<td>9,361,855</td>
<td>6.06e+08</td>
<td>9.36e+08</td>
<td>28,791.592</td>
<td>1.83e+04</td>
</tr>
</tbody>
</table>

Table VI. Performance of incremental verification

6. PERFORMANCE IMPROVEMENT USING EMBEDDDED C

The third verification step applies the embedded C constructs in PROMELA to improve model checking performance. The embedded C constructs were introduced to facilitate model-driven verification of software systems, making it possible to directly embed implementation code into PROMELA models [27]. This section explains how embedded C constructs are applied to existing Trampoline models and compares model checking performance before and after applying embedded C constructs.

As we already have Promela models for the Trampoline kernel, the use of embedded C constructs is not for embedding C code. Instead, we performed partial conversion of existing Trampoline models into embedded versions as follows:

1. Convert the atomic sequence of statements into `c_code` blocks.
2. Embed all global variables referenced/used from the converted `c_code` blocks into `c_code` blocks.
3. Embed all user-defined data types used in the `c_code` blocks into `c_decl` declarations.
4. Track each global variable declared in `c_code` blocks using the `c_track` construct.

Figure 8 shows a fragment of the Trampoline model converted from a pure Promela model into a model with embedded C constructs. The atomic sequence in the `inline` functions is converted into a `c_code` block, the global variable `tpl_fifo_rw` accessed from the `c_code` block is declared in a `c_code` block, and then the user-defined data type `TPL_FIFO_STATE` is declared in a `c_decl` block. Finally, the global variable is traced by using the `c_track` construct. Each `c_code` block is invoked during the model checking process, which is executed separately and returns its computation results. In this case, the model checker only needs to know the location and the size of the variable computed in the `c_code` block, regardless of how it is accessed or computed.

One thing worth noting is that each `c_code` construct is considered as a single transition and does not produce intermediate states in the model checking process, no matter how many statements are embedded inside, whereas each statement in the atomic sequence produces intermediate states in the pure PROMELA model. Considering that the cost for model checking grows linearly with the number of states and that the number of states tends to grow exponentially during the model checking process, this can result in a huge performance difference.

Table VII shows the performance data after applying embedded C constructs to the Trampoline kernel model. We see that both the absolute value of the verification costs and the rate of the cost increment as the number of API calls increases are greatly reduced. Figure 9 shows the comparative memory and time requirements for verifying the original Trampoline model and the model with embedded C; we note that the costs for the model with embedded C increase linearly as the number of API calls increases, whereas the original model shows an exponential cost increase.

We delayed the application of the embedded C constructs to the third step because merging transitions makes it difficult to analyze counterexamples and simulation results; the original
typedef struct TPL_FIFO_STATE{
    unsigned read;
    unsigned size;
} TPL_FIFO_STATE;

TPL_FIFO_STATE tpl_fifo_rw[n];

inline initialize_tpl_fifo_rw(){
    atomic{
        tpl_fifo_rw[0].read = 0;
        tpl_fifo_rw[0].size =0;
        tpl_fifo_rw[1].read =0;
        tpl_fifo_rw[1].size =0;
        ...
    }
}

Figure 8. Conversion example from the Trampoline OS

![Figure 8. Conversion example from the Trampoline OS](image)

<table>
<thead>
<tr>
<th>APIs</th>
<th>Depth</th>
<th>States</th>
<th>Transitions</th>
<th>Memory (Mbytes)</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>39,556</td>
<td>2.29e+06</td>
<td>2.50e+06</td>
<td>1,161.94</td>
<td>202</td>
</tr>
<tr>
<td>5</td>
<td>61,789</td>
<td>5.68e+06</td>
<td>6.29e+06</td>
<td>1,715.01</td>
<td>5.08e+02</td>
</tr>
<tr>
<td>7</td>
<td>78,724</td>
<td>9.44e+06</td>
<td>1.05e+07</td>
<td>2,118.33</td>
<td>8.46e+02</td>
</tr>
<tr>
<td>9</td>
<td>88,317</td>
<td>1.32e+07</td>
<td>1.48e+07</td>
<td>2,372.82</td>
<td>1.19e+03</td>
</tr>
<tr>
<td>11</td>
<td>96,095</td>
<td>1.70e+07</td>
<td>1.91e+07</td>
<td>2,886.24</td>
<td>1.54e+03</td>
</tr>
<tr>
<td>13</td>
<td>108,300</td>
<td>2.00e+07</td>
<td>2.24e+07</td>
<td>3,021.20</td>
<td>1.81e+03</td>
</tr>
<tr>
<td>15</td>
<td>121,007</td>
<td>2.46e+07</td>
<td>2.69e+07</td>
<td>3,284.29</td>
<td>2.17e+03</td>
</tr>
</tbody>
</table>

Table VII. Verification performance with embedded C

Figure 9. Comparison of verification performance

(a) Comparison on memory consumption

(b) Comparison on verification time

model is preferred for initial and incremental verification as long as the available resources allow this. Embedded C is the last choice for better scalability.

7. RELATED WORK

There have been a number of works on the formal verification of operating systems regarding various problems.
models and verifies the PATHO OS for vehicles using timed automata. Recent similar works are presented in [37, 38]. They model an OSEK/VDX application and the core part of its kernel in timed automata, and perform a rigorous timing analysis using the model checker Uppaal. Their modeling and verification focus on non-preemptive scheduling and an analysis of worst-case response time. It is subject to state-space explosion as the number of tasks increases because each task is explicitly modeled as an independent timed automata.

[32] suggest a meta-scheduler framework for implementing real-time scheduling algorithms. They present the design of the meta-scheduler and outline its implementation. The meta-scheduler is verified using Uppaal with respect to correctness, deadlock freedom, and livelock freedom. All these cases are mainly concerned with the abstract modeling of operating systems, but do not deal with implementation nor with the issue of scalability.

[8] is one of the typical approaches that model the memory map as new global variables in the embedded C source code and use a model checker to verify assertions. This approach does not take priority-based task scheduling into account. [20] presents a verification result for time partitioning in the DEOS scheduling kernel using the Spin model checker. It is the closest to our case study in its use of the Spin model checker and model translation from the kernel code. However, its verification is focused on one property regarding the scheduling algorithm so that aggressive abstraction techniques specialized for the property can be applied to effectively avoid the state-space explosion problem.

The Verisoft project [4] verifies runtime environment layers including OSEKtime and FlexRay, and application processes using the Isabelle theorem prover [18, 15]. The L4.verified project aims at providing consistency proofs for different layers of abstractions for the seL4 micro-kernel [19]. It takes the model-driven approach to develop a high-performance, low-complexity micro-kernel named seL4 from scratch, where the high-level micro-kernel model is specified in Haskell and is refined down to actual C code. The theorem prover Isabelle/HOL is used to verify the functional correctness of the micro-kernel and the consistency of the inter-layer functionality. However, the use of a theorem prover requires extensive knowledge about both the technique itself and the verification domain.

There have been a couple of approaches for model checking embedded software in general. [36] developed a special-purpose and domain-specific model checker named mc|square that verifies C code after it is compiled into Assembler. [21] verifies the embedded C code using an abstract memory model and then verifies software and hardware together by using only the return value from independently running hardware modules for static verification, which is similar, in principle, to the method supported by embedded C in PROMELA [27].

8. DISCUSSION

This work demonstrated an application of the model checking technique on the safety analysis of automotive software using the Trampoline operating system as a case example. We provided a conversion approach for the Trampoline kernel written in C into a formal model in PROMELA, a generic task modeling method for performing all possible interactions between tasks and the operating system, and a modeling approach for simulating a context-switching mechanism on a single processor. We believe that the suggested approaches are general enough to be applied to other embedded software running on a single processor machine. Nevertheless, one can still argue against the effectiveness of using formal methods in safety analysis for embedded software. This section discusses several related issues based on our experience from this study.

8.1. Why use SPIN?

Many existing approaches choose theorem proving for its thoroughness and soundness [15, 18, 19] for formal verification. However, the accessibility and usability of theorem proving are known to be lower than those of other automated verification techniques such as model checking and dynamic testing.
This work aims at providing a rigorous but efficient analysis technique that can be used by engineers without much knowledge regarding the theory of formal methods. SPIN is equipped with a visualized simulation and verification tool, which facilitates early assessment of the correctness of the model and assists in intuitive analysis through a visualized counterexample tracing mechanism. SPIN has been continuously evolved with enhanced techniques, such as swarm verification and model extraction [22, 27, 26], and is well supported with extensive experience on software model checking [9, 17, 10, 39, 20, 30]. These are our reasons for choosing SPIN as a primary verification tool.

8.2. Problems with existing C code model checking tools

There are a couple of well-known verification tools for C code. Most notably, CBMC [13], MODEX [28], and BLAST [6] have been well accepted and applied in C code verification. One of the benefits of these tools is that they can be directly applied to the source code without requiring conversion of the code into formal languages. CBMC and BLAST directly apply model checking techniques to the C code; MODEX automatically translates fragments of C code into PROMELA so that the modeling process can be eliminated. Nevertheless, this benefit is minimized when it comes to the verification of embedded software, where specifics of hardware-related controls and access need to be taken into account. We need to modify the C source code itself to reflect the hardware environment in order to be able to apply these language-specific model checking tools [36]. MODEX works quite well for translating small-size C source code into the PROMELA model, but the translation becomes error-prone when the size of the code gets larger and the code includes complex data structures. After an initial trial with those C code verifiers, we concluded that domain-specific model extraction, even though semi-automatic, can be much more efficient.

8.3. Testing vs. formal safety analysis

The safety problem identified in the Trampoline OS in this work might, in fact, have been identified using dynamic testing techniques if test cases were thoroughly defined. For example, we can drive exhaustive test cases based on all possible combinations of normal/abnormal values for each argument for each API. However, defining and executing such test cases for all possible combinations is quite costly. Trampoline defines more than 26 basic interfaces for system calls, where each of them has two arguments with 8-bit numeric types on average; this requires $26 \times 2 \times 2^8$ test cases for exhaustive testing. Even when we choose only boundary values for the arguments, at least $26 \times 2 \times 3$ test cases are required. The possible number of execution sequences for these $26 \times 2 \times 3$ test cases would rise to 156 factorial. Thus, model checking is not more expensive than testing when it comes to safety analysis.

8.4. Benefits of formal models

A notable benefit of having a formal model is that we can pre-check application designs before starting to code them. Like most other small-size embedded software, each application is compiled together with the Trampoline kernel code to generate an executable. This means that even a minor change in the application program requires recompilation and re-testing of the whole system. With formal models of the kernel code, we can perform arbitrary task behavior during the verification process and freely restrict the task behavior if a specific task sequence is to be simulated. Our models are designed in such a way that the generic task model can be simply replaced by a specific task design without affecting the kernel model so that model checking safety properties for a specific application is quite straightforward.

One can argue that the model extracted from the source code may miss important errors such as de-referencing null pointers, accessing out of array bounds, and failures in memory allocation that might reside in the source code. We suggest handling behavioral safety separately and independently from code safety; code safety, which includes the safe use of array indices and
pointer arithmetic, can be treated using static code analysis tools before handling behavioral safety issues.

9. CONCLUSION AND FUTURE WORK

We have presented our experience on model checking the Trampoline operating system for the purpose of safety analysis. To the best of our knowledge, this is the first extensive analysis of the Trampoline OS that has found an actual problem in the system.

Our approach can be differentiated from existing approaches in that the conversion and modeling of Trampoline into PROMELA faithfully preserve the original code except for some re-structuring and simplification of hardware-dependent code. This reduces the discrepancy between the original code and the model, thus minimizing accidental errors that might arise from the modeling process and providing straightforward counterexample replay in the actual code. However, this faithful translation naturally results in high complexity during verification. We anticipate that we will have to trade off either the accuracy of the kernel model or the generality of its environment for verification comprehensiveness. Experiments show that constraining the task model, which is the environment of the operating system, provides comprehensiveness up to a certain point, which is believed sufficient for automotive ECU controllers.

Though this work shows promising results that model checking may be engineered in such a way that it can be routinely applied for the safety analysis of automotive operating systems, it also includes pitfalls and room for improvement. First, the conversion from the Trampoline kernel to PROMELA is done manually with the aid of a code analysis tool, and thus includes potential human mistakes. To eliminate human mistakes, the manual model construction was thoroughly tested using the SPIN simulator and model checker, which in fact took more time than model construction itself†. The best way to avoid such manual construction and validation costs is to have the conversion process automated. We plan to develop a domain-specific model extraction tool based on our experience.

Second, model checking the Trampoline kernel involves several parameters other than the number of API calls per task, such as the number of activated tasks, the number of maximum activations per task, and the number of maximum resources/events. The model checking experiments in this work used fixed values for these parameters: 4 tasks with maximum of 2 multiple activations and resources/events per task. More refined experiments with varying parameters are necessary for a complete analysis. Since we expect that varying such parameters will result in increasing model checking complexity, future work would involve an investigation into developing a systematic method for reducing the complexity of the kernel model itself without requiring knowledge about implementation details.

ACKNOWLEDGMENT

This work largely benefited from prior work done by Seunghyun Yoon, Yeonjoon Kim, and Minkyu-Park, who performed SFTA for automotive operating systems in a related project and built the code extractor on top of the Understand tool.

REFERENCES

1. OSEK/VDX operating system specification 2.2.3.

†It took about 2 person-months to construct the initial model and about 3 person-months to validate the model in this case study.