IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 19, 2020, accepted July 9, 2020, date of publication July 22, 2020, date of current version August 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3011190

OS-Aware Interaction Model for the Verification of Multitasking Embedded Software

YUNJA CHOI
School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, South Korea
e-mail: yuchoi76@knu.ac.kr

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2016R1D1A3B01011685), and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science and ICT (NRF-2017M3C4A7068175).

ABSTRACT As the behavior of multitasking embedded software is dependent on the underlying operating system(s), rigorous and efficient verification in this domain requires models of operating systems (OS) that enable OS-aware verification of application programs at reduced cost. However, the heterogeneity of the languages used for OS models and of the program source code makes it difficult to compose these seemingly independent components and thus requires translation of one language into another, causing various issues in verification. To alleviate this problem, we propose a hybrid approach that composes formal OS models with application programs in an interaction model. Based on typical OS-application interaction behavior, our interaction model is a composition framework that connects an OS model to application programs as long as they share the same Application Program Interface (API). It provides seamless composition of two heterogeneous software artifacts by formulating source code annotations based on control-flow analysis and by synchronizing state transitions over API function calls to regulate the context switching of multitasking programs. A prototype implementation of the interaction model was applied to eight benchmark programs of the Erika OS and a control program with real-scale complexity from the automotive domain. It was shown that the framework supports systematic and effective verification of multitasking embedded software, which has not been possible using code-level model checking.

INDEX TERMS Embedded software, multi-tasking, heterogeneous composition, model checking.

I. INTRODUCTION

Embedded software[1] [15] controls hardware devices and typically runs on top of an operating system as the sole software on the target device. Some representative examples include ECU (Electronic Control Unit) control software in the automotive domain and control software for IoT (Internet of Things) devices, drones, and robots. The fact that, unlike typical general-purpose computer-based systems, the control software does not share operating system services with other applications makes it possible to optimize the software for memory efficiency, system safety, and performance at the cost of higher interdependency with its underlying operating system. Due to this close interdependency, however, verification of embedded software becomes more complicated, as the correctness or safety of the control program is inseparable from that of the operating system as well as from the interactions between them.

Nevertheless, formal verification approaches for multitasking embedded software have mostly focused on the control logic separately from the OS [1], [13], [30], [35], [38]–[40], [45], [47], which often produces a large number of false alarms due to the over-approximation of the environment, including the operating system. For example, the application program shown in Figure 1 has a unique execution trace due to the scheduling decision from its underlying operating system. However, if we treat the two tasks as being arbitrarily interleaved with each other without considering the behavior of the underlying OS, as most existing approaches do, there can be a maximum of ${}_{10}C_5 = 252$ possible execution traces, all of which but one are infeasible. It is difficult and inefficient to manually identify one true alarm out of a possible 252 alarms.

Anticipating the behavior of the underlying operating system may address this problem of verification accuracy, but

[1]This term is sometimes used interchangeably with firmware.

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

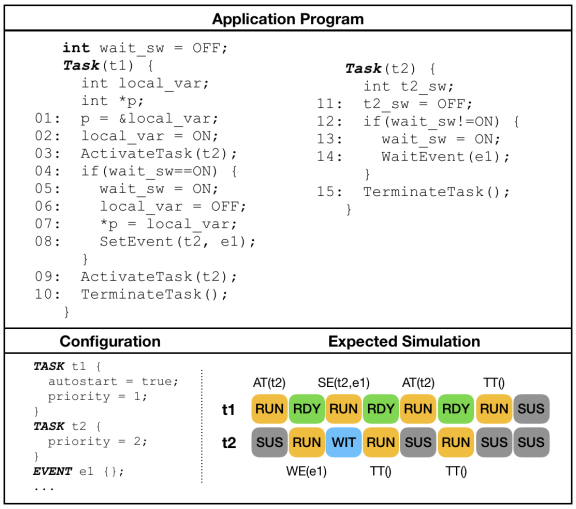

FIGURE 1. A multitasking program and its expected execution sequence.

has not been adopted as a practical solution due to the apparent increase of verification cost: The implementation of embedded OS typically comprises 4,000 to 10,000 lines of code, which is larger than the typical size of control software.

This work strives to improve the verification accuracy of multitasking embedded software while maintaining verification cost at a practical level, noting that formal models of operating systems can greatly help to reduce verification complexity [14], [43]. Assuming that the OS model is comprehensively verified with respect to functional correctness as well as system safety, we can perform two-step verification: (1) verification of the OS implementation using the OS model as a test oracle, and then (2) verification of the embedded software by replacing the OS implementation with the verified OS model. Such thoroughly verified operating system models already exist, for instance OSs developed using a proof-by-construction approach [5], [31] and formal OS models written in formal modeling languages [7], [26], [33], [52]. Among these, our previous approach [7] generates formal OS models by assembling predefined formal service patterns for a given system configuration, thereby providing a formal framework for the verification in accordance with the typical construction process of embedded software illustrated in Figure 2. The remaining issue is how to compose the generated formal OS models with control programs written in various programming languages.

To enhance the reuse of formal OS models and the composability of OS models and application programs, this work introduces a framework consisting of an interaction model between an OS model and the control software and a method for embedding application programs into the interaction model. Our framework considers an OS model as a black box that interacts with application programs only through APIs and can thus be applied to various OS models as long as they are designed to react to API function calls and allow references to internal system variables, such as internal states of tasks or values of events handled by the kernel.

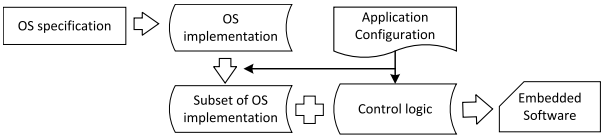

FIGURE 2. Construction of embedded software.

To embed the application source code into the interaction model, we define an application wrapper that incorporates a synchronization mechanism for a visible state, i.e., a state that calls API functions or references global variables, into the program source code. The synchronization mechanism is used to control the granularity of the context switching among tasks with minimum changes to the application program. It is shown that the application wrapper is a sound abstraction of the program source code if there is no race condition in the program.

As our framework is designed to be independent of OS models, we do not assume any specific scheduling algorithms or modeling languages. To demonstrate the effectiveness of our approach, however, we implemented the framework using Promela [25] as the target modeling language. The effectiveness of the framework is demonstrated through applications on eight benchmark programs for the Erika OS [17] and a control program with real-scale complexity from the automotive domain,[2] showing that the approach reduces verification cost while improving verification accuracy.

The major contributions of this work can be summarized as follows

- It formally defines an interaction model and an application wrapper that enable heterogeneous composition of a model of the operating system and the source code of embedded control software. To the best of our knowledge, this is the first work addressing the seamless composition of these two heterogeneous software artifacts for verification purposes.
- Comparative experiments with existing state-of-art verification approaches on a set of benchmark programs for Erika OS [17] and a representative embedded control software (Winlift [34]) in the automotive domain demonstrate that the use of this interaction model improves verification accuracy and is more scalable, outperforming the state-of-art verification tools.
- Accurate verification of multitasking embedded software becomes feasible with moderate cost, which has not been possible using existing approaches.

The remainder of this paper is organized as follows: Section II provides a brief introduction to embedded software. Section III and Section IV define the OS-aware interaction model and the notion of the application wrapper, respectively. Section V describes how to construct an interaction model using Promela as a case example. Section VI shows the application of the interaction model to

[2]We chose these programs because they are publicly available. It is rare to find real-scale open-source programs in this domain.

a set of control programs. After summarizing related work in Section VII, we conclude with a discussion in Section VIII.

II. BACKGROUND

A. ELEMENTS OF EMBEDDED SOFTWARE

Embedded software controls hardware devices and typically runs on top of an operating system as the sole software on the target device. Some representative examples include ECU control software in the automotive domain and control software for IoT devices such as drones and smart watches. The fact that, unlike typical general-purpose computer-based systems, the control software does not share operating system services with other applications makes it possible to optimize the software for memory efficiency, system safety, and performance at the cost of higher interdependency with its underlying operating system.

Figure 2 illustrates a typical construction process of embedded software. A subset of an (in-house or commercial) OS implementation is selected according to the system configuration and is compiled together with the application control logic to generate the target embedded software. The control program is typically multitasking, consisting of a set of tasks (or threads). Each task has its own priority so that a task with higher priority is scheduled earlier than tasks with lower priority. It is also possible for a task with lower priority to run prior to a task with higher priority if it accesses a critical section by occupying resources. A running task may be in a waiting state by voluntarily waiting for an event. An embedded operating system such as Erika [17], Zephyr [18], or FreeRTOS [19] typically maintains four internal states of each task, {running, ready, waiting, suspended}, with minor variations. A control program interacts with its underlying operating system through API functions provided by the OS.

B. VERIFICATION ISSUES

Simulation or Hardware-in-the-Loop(HiL) testing are typical verification methods used in practice in this domain, but the quality of the simulation or HiL testing largely depends on the quality of the selected input values. These are known to be insufficient for comprehensive verification if the goal is to identify subtle problems in critical systems. Formal verification approaches for control logic have been investigated since the 1990s to address this issue. However, formalizing only the control logic has not been sufficient for effective verification, as it results in producing a large number of false alarms due to the over-approximation of environments, including operating systems. Formal modeling and verification of embedded systems has been actively investigated, but research is still considered to be at too early a stage to be practical.

Figure 1 shows an example of an embedded application program operated in an automotive operating system that is compliant with the OSEK/VDX international standard for road vehicles [12]. Given the system configuration, the two tasks are expected to be executed as shown in the lower right part of the figure, as the OSEK/VDX OS adopts

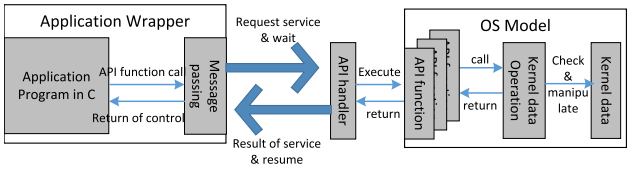

FIGURE 3. Schematic view of the interaction model.

a priority-based FIFO (First-In-First-Out) scheduling algorithm. The autostart task t1 runs first and activates task t2, which has higher priority. Task t2 preempts t1, runs to set *wait_sw* to *ON*, and goes to the waiting state by calling *WaitEvent(e1)*, giving another execution round to task t1. t1 calls *SetEvent* for task t2, as the branch condition in line 04 evaluates to true, which preempts t1 again and terminates. Task t1 activates t2 again, but terminates immediately as *wait_sw* evaluates to *ON*.

The decision on which task gets CPU (Central Processing Unit) time cannot be described in a simple scheduling algorithm, as it depends on task priorities, API call sequences, resource allocation, interrupts, events and alarms, and many other aspects of the system. If we try to verify this multitasking program without considering its underlying operating system or try to do so with a highly abstracted operating system, e.g., allowing non-deterministic scheduling, formal verification has to consider all possible execution traces for the same program, as context switching may occur at any instruction of each task. This is extremely expensive both in terms of performing the formal verification and in terms of identifying false alarms produced by over-approximated OS behavior.

III. OS-AWARE INTERACTION MODEL

This section introduces an OS-aware interaction model as a composition framework for OS models and application source code written in different languages. Figure 3 shows an overview of the interaction model. The interaction model consists of an OS model and an application wrapper containing the target application program to be verified. The OS model typically consists of kernel data, kernel operation, a set of API functions, and API handlers. The target application program is embedded into the application wrapper, which delivers messages between the API handler of the OS model and the application program being executed.

Definition 1: An embedded software M is a parallel composition of an embedded operating system M_{os} and an application software M_{app}, synchronized over the set of API function calls E:[3]

$$M = M_{os} || M_{app},$$

where M_{os} and M_{app} are represented as parameterized statemachines[4] as defined in Definition 2 and Definition 4.

[3]Synchronized parallel composition allows each statemachine to accept a transition while the others remain in the same state, but all statemachines satisfying the transition conditions must undergo the transitions at the same time.

[4]Each statemachine is parameterized with the system configuration and utilized as a statemachine pattern [7].

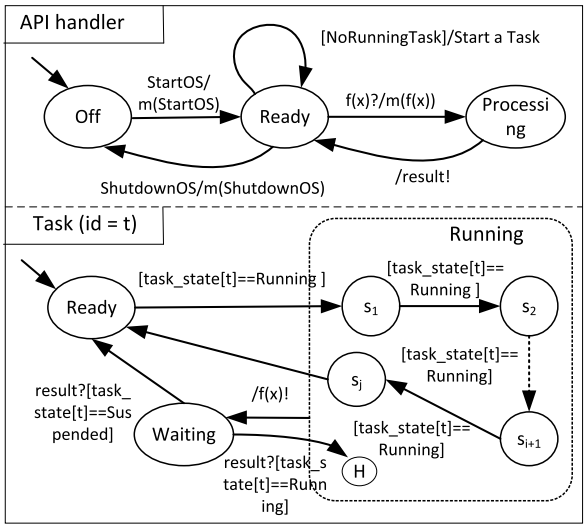

FIGURE 4. A model of an API handler and a task.

An embedded operating system typically consists of a kernel for core system services, ISR handlers, and alarm handlers. Therefore, $M_{os} = M_{kernel}||M_{alarm}||M_{isr}$ is a parallel composition of OS kernel, periodic alarms, and interrupt handlers. However, our interaction model abstracts M_{os} with M_{ah}, a model for API handlers, as we consider the OS as a black box and focus on external interactions with application programs.

Definition 2: An API handler $M_{ah} = (S, V, E, A, R, I)$, an abstract representation of an embedded operating system M_{os}, is a statemachine reacting to calls to a predefined set of API functions, where

- $S = \{Off, Ready, Processing\}$ is a set of system states,
- V is a set of system variables (called *visible* variables),
- $E = (\Sigma \cup \{\tau\}) \times P^*$ is a set of API function calls, where Σ is the set of names of API functions, τ represents no function call, and P is a set of parameters,
- A is a set of action sequences with an action map $m : \Sigma \times P^* \longrightarrow A \times P^*$,
- $R \subseteq S \times E \times C[V] \times A \times S$ is a set of transitions, where $C[V]$ is a set of conditions over V, and
- $I = \{Off\}$ is a set of initial states.

The top part of Figure 4 graphically illustrates M_{ah} with three major states, *Off*, *Ready*, and *Processing*. Once it starts, it is in the *Ready* state and waits for external events such as an API function call from a task, then moves to the *Processing* state as it calls the processing functions $m(f(x))$ corresponding to the externally called API function $f(x)$. After the processing of $m(f(x))$, it sends out the result of the processing and goes back to the *Ready* state. The *Processing* state is highly abstracted in this figure, but it subsumes all the activities the kernel performs, such as task management, task scheduling, resource management, event management, and alarm/interrupt handling. Especially the system variables, such as the internal states of each task, are updated only by M_{os}, which is hidden from the scene.

Multitasking application software consists of a parallel composition of tasks, where each task shares global variables and communicates with its underlying operating system through API function calls.

Definition 3: $M_{app} = T_1||T_2||\ldots||T_n$ is a parallel composition of a finite number of tasks $T_k = (N_k, R_k, \{n_k^0\}, \{n_k^t\}, V_k)$, where

- $N_k \supseteq E$ is a set of nodes; each statement in a task is considered as a node, including API function calls and assignments,
- $R_k \subseteq N_k \times N_k$ is a set of directed control flow edges,
- V_k is a set of variables used in N_k, including global variables, and
- n_k^0 and n_k^t are the initial and the terminal node, respectively.

A node $n \in N_k$ represents a statement in the task and an edge $r \in R_k$ represents a control transition between two statements. A unique initial and terminal node is assumed in each task as multiple terminal nodes can be directed to one terminal node by adding an edge between them.

As each task in an application program has its own control flow structure, M_{app} could have arbitrary interleavings among multiple tasks unless it is constrained by the operating system. In our interaction model, the original application program M_{app} is replaced with the following abstract representation:

Definition 4: $\widetilde{M_{app}} = M_1||M_2||\ldots||M_n$ is a parallel composition of a finite number of tasks $M_k = (S_k, V_k, E, R_k, \{s_1\}, \{s_f\})$, where

- $S_k = \{Ready, Waiting\} \cup (Running = \{s_i\}_{i \in N})$ is a set of states, where *Ready* is the initial state of M_k;
- E is the same set of API function calls as in M_{ah};
- $R_k \subseteq S_k \times E \times C[V_k] \times S_k$ is a set of transitions, where $C[V_k]$ is a set of conditions over V_k; and
- s_1 and s_f are the initial and the final sub-states of *Running*, respectively.

Each task M_k consists of three major states: *Ready, Waiting, and Running*. The *Running* state is further decomposed into multiple sub-states $\{s_i\}$, where each state s_i is a group of multiple statements that can be executed atomically. The lower part of Figure 4 shows a fragment of M_t for a task with id t. Once it starts, it goes to the *Ready* state, waiting for the kernel scheduler to change its state to *Running*. If its internal state assigned by the OS kernel becomes *Running*, it goes to the initial sub-state s_1 of the *Running* state. Transitions among the sub-states follow the control flow of the task except for the case when the task calls an API function that causes a transition to the *Waiting* state. If the task receives the result and its internal state assigned by the OS remains *Running*, it goes back to the previous sub-state of *Running*. The `H` sign inside the *Running* state in Figure 4 represents that it remembers the last sub-state before exiting the state. If the result of the API call changes the internal state of the task to *Suspended* (which is the case when the API call is for terminating the task), it goes back to the *Ready* state. Here, *task_state[t]* is a shared system variable whose values can be updated only by the OS kernel. Each state transition within the *Running* state is guarded by `[task_state[t]==Running]`, meaning

that the transition occurs only if the internal state of the task is `Running`. The task exits from *Running* to *Ready* after the final sub-state of *Running*.

We note that the model does not explicitly deal with cases caused by external interrupts. Interrupts are handled by the OS and may change the internal state of each task as a result. The task in the *Running* state waits in a sub-state if an interrupt changes its internal state and resumes as soon as the guarding condition to transit to the next sub-state becomes true.

With this interaction model in mind, the subsequent sections will show how M_{app} is converted into $\widetilde{M_{app}}$ without losing its soundness.

IV. APPLICATION WRAPPER

In order to embed the application source code into the interaction model with minimal modification, we first refine the notion of control flow of each task and define the grouping rules of the nodes to separate visible nodes from invisible nodes.

A. EXTENDED CONTROL FLOW REPRESENTATION

Definition 5: Given a task T in an application program,[5] a control flow structure $T_{CFG} = (N, R, \{n_0\}, \{n_t\}, V)$ is extended from Definition 3 by refining N and V as follows:

- $N : N_a \cup N_c \cup N_{API} \cup N_b \cup N_p$ is a set of nodes where N_a is a set of declaration/assignment nodes, N_c is a set of application function call nodes, N_{API} is a set of API function call nodes, N_b is a set of branch nodes, and N_p is a set of pseudo nodes;
- $V = V_v \cup V_{iv}$ is a set of (visible/invisible) variables used in N.

Especially the set of variables V is categorized into two parts: *visible* variables and *invisible* variables. A visible variable is a global variable or system variable. An invisible variable is a local variable used within a task. We represent an immediate successor m of a node n as $n \Rightarrow m$ and use $n_0 n_1 n_2 \ldots n_k$ for a sequence of nodes that satisfies $n_i \Rightarrow n_{i+1}$, $\forall i = 1..k-1$. The lower part of Figure 5 shows an example of the control flow structure visualized for a typical task written in C, namely the Producer task shown in the upper part of the figure.

A branch node is related to a sequence of nodes to be executed when the branch condition is true (the true block) and a sequence of nodes to be executed when the branch condition is false (the false block). These blocks can be characterized as their first nodes are successors of the given branch node and their last nodes are predecessors of the same node that is the start node after the branch statement. The transitions from a branch node n_b to the first node m of the true block and to the first node l of the false block are denoted as $n_b \Rightarrow_t m$ and $n_b \Rightarrow_f l$, respectively.

[5]We assume that a task is distinguishable from ordinary functions, e.g., by using the keyword *Task*, which is a typical case in embedded software.

Definition 6: Given a branch node $n_b \in N_b$, a true block of n_b, $T(n_b) = n_0 n_1 n_2 \ldots n_m$ and a false block of n_b, $F(n_b) = n'_0 n'_1 n'_2 \ldots n'_l$ are sequences of nodes, where

- $n_b \Rightarrow_t n_0$ and $n_i \Rightarrow n_{i+1}$, $\forall i = 0..m-1$,
- $n_b \Rightarrow_f n'_0$ and $n'_i \Rightarrow n'_{i+1}$, $\forall i = 0..l-1$, and
- $\exists! n_k \in N$ such that $n_m \Rightarrow n_k$ and $n'_l \Rightarrow n_k$.

B. STATEMACHINE REPRESENTATION OF A TASK

The extended control flow structure of a task is mapped to a statemachine representation in order to construct an application wrapper. The application wrapper identifies API function call nodes in compound statements, such as loop statements and conditional statements, as a call may cause rescheduling of tasks and change the execution sequence of the multitasking program. For example, calling `WaitSem(p)` inside the while loop in Figure 5 may cause control being handed over to the OS and having to wait for the result before proceeding to execute the next statement. In addition, nodes containing visible variables get special treatment, as the order of assignments/references to these variables may have a critical impact on the behavior of concurrent programs. The context switch tester identifies such *influential* context switching points.

Definition 7: A context switch tester

$$Test_{cs} : N \longrightarrow \{T(N_b), F(N_b), False, True\}$$

is a function over a set of nodes that returns

- *True*, if n contains an API function call or visible variables, or if $n \in N_b$ and $\exists m \in T(n)$, $\exists k \in F(n)$ such that $Test_{cs}(m) = Test_{cs}(k) = True$,
- *False*, if n does not contain any API function call nor any visible variables, and $n \in N_b$ implies $Test_{cs}(m) = False$, $\forall m \in T(n) \cup F(n)$.
- $T(n)$, if $n \in N_b$ and $\exists m \in T(n)$ such that $Test_{cs}(m) = True$, but $Test_{cs}(m) = False$, $\forall m \in F(n)$,
- $F(n)$, if $n \in N_b$ and $Test_{cs}(m) = False$, $\forall m \in T(n)$, but $\exists m \in F(n)$ such that $Test_{cs}(m) = True$

The application wrapper constructs an abstract state out of a group of consecutive control flow nodes, depending on whether or not there are any API function calls and visible variables in the nodes.

Definition 8: For a given task $t = (N, R, n_0, n_t, V)$, a statemachine wrapper (or an application wrapper) $M_t = (S, s_0, s_t, \hat{R})$ is defined as

- $s_0 = s_t = \{Ready\}$,
- S is a set of states whose elements are determined by the grouping function $g : N \longrightarrow S$ that satisfies the following relations:
  1) $g(n_0) = g(n_t) = Ready$,
  2) $\forall n, m \in N$ such that $n \Rightarrow m$, $(Test_{cs}(n) = False \wedge Test_{cs}(m) = False)$ iff $g(n) = g(m)$,
  3) $\forall n \in N_b$ such that $Test_{cs}(n) \neq False$, $g(m) \neq g(k)$, $\forall m \in T(n)$, $\forall k \in F(n)$,
- $\hat{R} \subseteq S \times C \times S$ is a set of transitions, where C is a set of predicates over V and is defined as follows:
  1) $\forall s \in S$, $(s, True, s) \in \hat{R}$, i.e., $\hat{R}$ includes self-transition,

2) $\forall s, s' \in S$, $(\exists n \in s, \exists m \in s'$ such that $n \Rightarrow m)$ implies $(s, True, s') \in \hat{R}$,
3) For $s = g(n_{api}) \in S$, $(n_{api}, m) \in R$ implies $(s, True, Waiting) \in \hat{R}$ and $(Waiting, runnable[t], g(m)) \in \hat{R}$,
4) For $n \in N_b$ where $Test_{cs}(n) \neq False$, $m \Rightarrow n \wedge n \Rightarrow_t l$ implies $(g(m), n.condition, g(l)) \in R_{temp}$,
5) For $n \in N_b$ where $Test_{cs}(n) \neq False$, $m \Rightarrow n \wedge n \Rightarrow_f l$ implies $(g(m), \neg n.condition, g(l)) \in R_{temp}$, and
6) $\{(s_i, \bigwedge_k c_k, s_j) \mid (s_i, c_k, s_j) \in R_{temp}\} \subset \hat{R}$.

Intuitively, the statement wrapper defined in Definition 8 can be explained as follows: To define states in the statemachine wrapper, 1) both the initial and the final state in the wrapper are *Ready*, 2) two assignment nodes n and m are grouped into the same state, i.e., $g(n) = g(m)$, if and only if $m \Rightarrow n$ or $n \Rightarrow m$ and they do not contain API function calls or visible variables, and 3) any nodes in the true block of a branch node n_b cannot be grouped into the same state to which a node in the false block of the same branch node belongs if $Test_{cs}(n_b) \neq False$.

To define transitions in the statemachine wrapper, 1) self-transition is included for each state, 2) two states s and s' have a transition relation if there is a node n from s and a node m from s' with an edge $n \Rightarrow m$ in the control flow structure, and 3) a transition from an API call node n_{api} to a node m is converted into a transition from $s \in g(n_{api})$ to *Waiting* and a transition from *Waiting* to $g(m)$. The last three rules are for defining transitions from branch nodes. 4) If a branch node n has potential switching points, the condition of the branch condition becomes the transition condition between $g(m)$ and $g(l)$, where m is the predecessor node of n and l is the first node of its true block; likewise, 5) the negation of the branch condition becomes the transition condition between $g(m)$ and $g(l)$, where m is the predecessor node of n and l is the first node of its false block. The final transition condition between $g(m)$ and $g(l)$ is the conjunction of all transition conditions identified from 4) or 5).

For example, as shown in the lower part of Figure 5, the control flow structure is first annotated with n_p, n_c, n_a, n_b, and n_{api}, depending on the type of each node. The second round performs $Test_{cs}$ for each branch node and annotates it as n_{api} if $Test_{API}(n_b) \neq False$, meaning that the branch node is followed by an API function call or assignments/references to visible variables and thus needs to allow a context switch within the branch structure. As the result of the second-round annotation, a branch node is either embedded into an invisible state or used as a transition condition between its visible predecessors and successors. The branch node "EE_TRUE" is now annotated with n_{api} after the second-round annotation in Figure 5. Figure 6 shows the statemachine representation after application of the grouping rules to the annotated control flow structure.

We assume atomic execution of the statements in each state in the wrapper statemachine; i.e., no interrupt is allowed while executing statements in a state. We also assume that the

```
TASK(Producer) {
  volatile int i;
  static int pcounter = 0;
  while (EE_TRUE) {
     task1_fired++;
     pcounter++;
     printf("Producer before WaitSem P, %d\n", pcounter);
     if (pcounter == 2) {
          EE_assert(EE_ASSERT_TASKP_WAIT, pcounter == 2,  EE_ASSERT_INIT);
     }
     WaitSem(P);
     /* take some time to produce an item */
     for (i = 0; i < PRODUCTION_CYCLES; i++);
     printf("Item produced, before PostSem V\n");
     if (pcounter == 2 ) {
          EE_assert(EE_ASSERT_TASKP_POST, pcounter==2, EE_ASSERT_TASKC_POST);
     }
     PostSem(V);
     printf("Item produced, after PostSem V\n");
     if ( pcounter == 3 )  break;
  }
  TerminateTask();
}
```

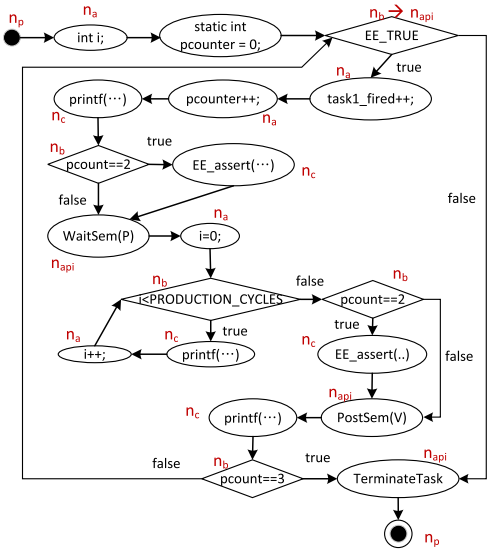

FIGURE 5. A sample multitasking program and its annotated control flow structure.

`runnability` of the task, which is determined by the OS scheduler, is checked before the transition to another state. If the task is not runnable, it stays in the same state until it becomes runnable:

$$\frac{t.state == p = \{n_1 n_2 n_3 \ldots n_k\},\ t.runnable,\ (p, c, q) \in t.\hat{R},\ c}{atomic\{n_1.statement; \ldots; n_k.statement;\}, t.state = q},$$

$$\frac{t.state == p = \{n_1 n_2 n_3 \ldots n_k\}, \neg t.runnable \vee \neg c, \forall (p, c, q) \in t\hat{R}}{t.state = p}$$

C. SOUNDNESS

Assuming that the OS model is a sound abstraction of all possible implementations of the functional requirements of the OS, the soundness of the interaction model depends on the soundness of the application wrapper. We show this soundness using an application with two tasks. Generalization to arbitrary n tasks can be done in a similar way.

Theorem 1: For a given $M_{app} = T_1||T_2 = (N_1 \times N_2, R_1 \times R_2, (n_1^0, n_2^0), (n_1^t, n_2^t), V_1 \cup V_2)$, let

- $\widetilde{M_{app}} = M_{T_1}||M_{T_2} = (S_1 \times S_2, \hat{R}_1 \times \hat{R}_2, (s_1^0, s_2^0), (s_1^t, s_2^t), V = V_1 \cup V_2)$,

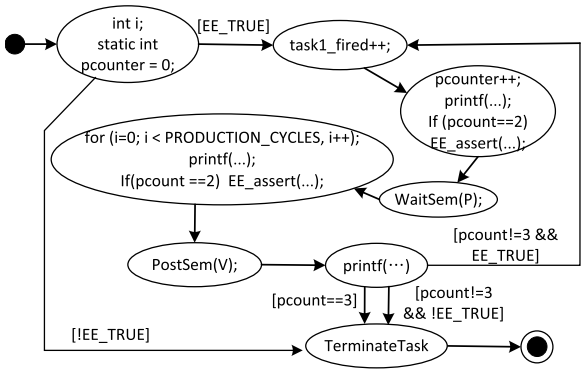

FIGURE 6. Statemachine representation of the sample control flow structure in Figure 5.

- $\alpha = g \times g : N_1 \times N_2 \rightarrow S_1 \times S_2$ be an abstraction function, where $\alpha(n_1, n_2)$ is defined as $(g(n_1), g(n_2))$,
- $L : N_1 \times N_2 \rightarrow 2^V$ be a labeling function for M_{app}, a valuation function for all visible variables in a node pair,
- $\hat{L} : S_1 \times S_2 \rightarrow 2^V$ be a labeling function for $\widetilde{M_{app}}$, a valuation function for all visible variables in a state pair;
- $\preceq$ represent the abstraction relation over $((N_1 \times N_2) \times (S_1 \times S_2))$such that $(n_1, n_2) \preceq (s_1, s_2)$ if $(s_1, s_2) = \alpha((n_1, n_2))$,

then $\preceq$ is a simulation preorder if there is no race condition in M_{app}.

Proof: It suffices to show that (1) all initial states in M_{app} are mapped to initial states in $\widetilde{M_{app}}$, (2) all transitions in M_{app} are mapped to transitions in $\widetilde{M_{app}}$, and (3) the label is the same before and after the mapping.

1) As we assumed a unique initial node in a task, there is a unique initial node, which is $(n_0^1, n_0^2) \in N_1 \times N_2$. Then $\alpha(n) = (g(n_0^1), g(n_0^2)) = (Ready, Ready)$ is an initial state of $\widetilde{M_{app}}$ through the construction of $\widetilde{M_{app}}$.
2) Let $p = (n_1, n_2) \in N_1 \times N_2$ such that $\alpha(p) = (s_1, s_2) \in S_1 \times S_2$. For any $q = (n_1', n_2')$ such that $(p, q) \in R_1 \times R_2$, $(n_1, n_1') \in R_1$ and $(n_2, n_2') \in R_2$ by the definition of the transition relation. Then $(g(n_1), g(n_1')) \in \hat{R}_1$ and $(g(n_2), g(n_2')) \in \hat{R}_2$ according to the second rule of $\hat{R}$ in Definition 8. Therefore, $\alpha((p, q)) = (g(p), g(q)) \in \hat{R}_1 \times \hat{R}_2$.
3) Let $n = (n_1, n_2) \in N_1 \times N_2$ with $\alpha(n) = (g(n_1), g(n_2))$. If n_i contains a visible variable, then $g(n_i) = \{n_i\}$ according to the construction of $\widetilde{M_{app}}$. If neither n_1 nor n_2 contains a visible variable, then neither $g(n_1)$ nor $g(n_2)$ can contain visible variables as g groups nodes with visible variables and nodes with invisible variables separately. Therefore, $L_i(n_i) = \hat{L}_i(g(n_i))$ for both cases, where $L_i : N_i \rightarrow 2^V$ and $\hat{L}_i : S_i \rightarrow 2^V$ are partial valuation functions. Finally, if there are no race conditions in M_{app}, we can safely say that $L_1(n_1) = L_2(n_2)$ and $\hat{L}_1(s_1) = \hat{L}_2(s_2)$, $\forall(n_1, n_2) \in N_1 \times N_2$ and $\forall(s_1, s_2) \in S_1 \times S_2$, i.e., the values of the visible variables are consistent in each state. Therefore, $L((n_1, n_2)) = \hat{L}(\alpha((n_1, n_2)))$, $\forall(n_1, n_2) \in N_1 \times N_2$.

Therefore, $\widetilde{M_{app}}$ is a sound abstraction of M_{app}.

V. CONSTRUCTION

The interaction model can be realized in many different ways as the OS model can be constructed in any formal modeling language. Nevertheless, this section explains the construction of an interaction model assuming the use of Promela as the modeling language since (1) due to its C-like syntax, it is easier to understand for readers who are unfamiliar with formal modeling languages, and (2) its support for C code embedding is straightforward to understand and to use.

A. OVERVIEW OF PROMELA

Promela is the modeling language for Spin [25]. The syntax of Promela is similar to the C language, but its semantics is based on CSP (Communicating Sequential Processes) [24]. Major constructs of Promela include `proctype`, which defines the type of a process, and `chan`, which defines a communication channel used to pass messages among processes. The upper right part of Figure 7 shows a process that communicates through the channel `api_ch`. Each statement in Promela is executed only if the evaluation of the statement is true. For example, `api_ch?[_,eval(StartOS),_,_]` is executed when the `api_ch` channel receives a message where the value of the second field of the message is equal to `StartOS`.

There are two major constructs for embedding C source code into Promela: `c_code` and `c_expr`. Statements inside a `c_code` block are executed according to the control flow, but value changes are not traced by the model checker. `c_expr` is used to evaluate expressions in the C code embedded in the model. We can use these constructs to execute a portion of a C program and get the execution result without keeping track of uninteresting variables. It is also possible to trace a specific variable used inside a `c_code` block by using a `c_track` construct, meaning that we can control the level of abstraction of the model by tracing only those values that are relevant for the verification goal [48].

B. INTERACTION MODEL IN PROMELA

As explained in the previous sections, we represent embedded software as a composition of M_{ah} and $\widetilde{M_{app}}$. M_{ah} is independent of the application software and can thus be predefined and reused as a Promela model for a given set of API functions. The statemachine model of the API handler in the upper part of Figure 4 is realized with the `proctype` construct as shown in the upper right part of Figure 7: It waits for messages coming through the message channel `api_ch`. Once a message arrives, it identifies the API function corresponding to the message, removes the message from the channel, and performs the corresponding services specified in the OS model.

$\widetilde{M_{app}}$ is application-specific and needs to be constructed from the program code. Therefore, only a skeleton of each task in the application program is declared as an independent

```
proctype App(byte tid){
  main :
    if
    :: tid == 0 -> goto task0;
    :: tid == 2 -> goto task2;
    :: tid == 1 -> goto task1;
    :: ...
    fi;

task0:
    TASK_Task0(tid);
    goto task0;

task2:
    TASK_Producer(tid);
    goto task2;

task1:
    TASK_Consumer(tid);
    goto task1;

...
}
```

```
proctype API_handler(){
  // declaration of local variables
  Off:
   api_ch?[_,eval(StartOS),_,_];  api_ch?_,_,_,_;
   goto Ready;
  Ready:
   if
   :: d_step{ api_ch?[_,eval(ActivateTask),_,_]
      -> api_ch?param0,_,param1,_; }
         activate_task(param0,param1);
         api_ch!0,RT,param0,0;
   :: d_step{ api_ch?[_,eval(Schedule),_,_]
     -> api_ch?param0,_,_,_; }
         schedule(param0);
         api_ch!0,RT,param0,0;
   ::  ...
   :: d_step{ api_ch?[_,eval(ShutdownOS),_,_]
     -> api_ch?param0,_,_,_; }
         shutdown(param0);
         api_ch!0,RT,param0,0;
         goto Off;
   fi;
  goto Ready;
}
```

```
inline TASK_Producer(tid){
  task_state[tid] == Running;
  do
  :: c_expr{EE_TRUE} ->c_code{ task1_fired++; };
                       task_state[tid] == Running;
                       c_code{ pcounter++;  printf( ...);
                                if(pcounter == 2){ EE_assert(...); }
                       };
                       task_state[tid] == Running;
                       api_ch!tid,WaitSem,task[tid].dyn_prio,P;
                       api_ch?[_,eval(RT),eval(tid),_];   api_ch?_,_,_,_;
                       task_state[tid] == Running;
                       c_code{
                          for (i = 0; i < PRODUCTION_CYCLES; i++)
                               printf(...);
                          if (pcounter == 2 ) { EE_assert(...); }
                       };
                       task_state[tid] == Running;
                       api_ch!tid,PostSem,0,V;
                       api_ch?[_,eval(RT),eval(tid),_];   api_ch?_,_,_,_;
                       task_state[tid] == Running;
                       if
                       :: c_expr{pcounter == 3} ->      break;
                       :: else -> skip;
                       fi;
  ::else -> break;
  od;
 task_state[tid] == Running;
 api_ch!tid,TerminateTask,0,0;
 api_ch?[_,eval(RT),eval(tid),_]; api_ch?_,_,_,_;
}
```

FIGURE 7. API handler and application wrapper in Promela.

process, as shown in the upper left part of Figure 7. A process is activated by running the proctype `App` with a specific `tid`. In Promela, any activated process is runnable, but we add explicit runnability checking in the inline function for each task whose body is defined by the application wrapper. The lower part of Figure 7 shows the inline function `TASK_Producer` for embedding the task `Producer` shown in the upper part of Figure 5. The first `while` loop is converted into the Promela `do .. od;` loop because the true block of the branch node contains API function calls, `WaitSem` and `PostSem`. The branch condition `EE_TRUE` is checked within `c_expr` as it is declared in C source code and the Promela model is not aware of its existence. The inside of a true/false block may consist of several visible states, interaction points sending and receiving messages

TABLE 1. Mapping from wrapper states to Promela.

Statemachine wrapper	Promela construct
An invisible state $s = \{n_1 n_2 \dots n_k\}$	RUNCHECK; c_code{ n_1.statement; n_2.statement; ...; n_k.statement; };
A visible state $s = \{n_{api}\}$	RUNCHECK; APICALL; RETURN;
A visible state $s = \{n_a\}$	RUNCHECK; n_a.statement;
A transition condition C for loop branch n_b	RUNCHECK; do :: c_expr{C} $\rightarrow$ conversion of $g(T(n_b))$ :: else $\rightarrow$ break; od;
A transition condition C for conditional branch n_b	RUNCHECK; if :: c_expr{C} $\rightarrow$ conversion of $g(T(n_b))$ :: else $\rightarrow$ conversion of $g(F(n_b))$ fi;

to/from the OS model, and hybrid statements that determine the controls of the model by evaluating branch conditions using `c_expr`. In between these states, a guarding condition `task_state[tid] == Running` is inserted, which allows the execution of the state blocks only when the task is in the running state.

Table 1 shows the mapping rule from the statemachine wrapper for each task to the Promela constructs for embedding the source code within the interaction model. Here, `RUNCHECK` is used for checking the runnability of the task by checking whether the internal state of the task (updated by the OS) is in the `Running` state. If it is, the task continues to execute the next statement; otherwise, the task is blocked until it becomes runnable. Therefore, `RUNCHECK` is a potential context switch point among tasks. APICALL and RETURN represent making an API function call through a message channel and sending a return message through the message channel, respectively.

C. IMPLEMENTATION OF A PROTOTYPE TOOL

We implemented a prototype tool for the construction of the OS-aware interaction model from four software artifacts: the system configuration, the OS patterns, abstracted platform-dependent library functions, and an application program, as illustrated in the gray boxes in Figure 8.

The system configuration and the OS patterns are used for constructing configuration-dependent OS models. We reused the prototype tool developed for the pattern-based OS model generation framework [7] for the OS model generation part. The platform-dependent library functions used by the application program were abstracted manually to make them platform-independent. The application program was parsed and analyzed for constructing the Control Flow Graphs (CFGs), which are annotated with the types of nodes. We used EclipseCDT for parsing and analyzing the CFGs. The annotated CFGs were converted into an application wrapper. The prototype tool generates the wrapper and composes it with the generated OS model.

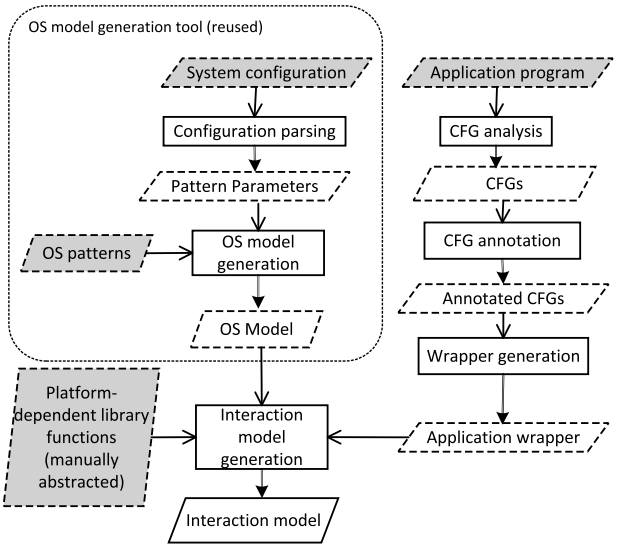

FIGURE 8. Construction of interaction model.

VI. EVALUATION

We conducted three sets of case experiments using the prototype tool to answer the following questions:

- RQ1 Is the proposed approach capable of identifying subtle issues in the verification of small-scale embedded software?
- RQ2 Is the proposed approach applicable to realistic embedded control programs?
- RQ3 Does the proposed approach perform better than C code model checking?

To answer these questions, we performed three sets of experiments on: (1) eight benchmark programs of the Erika OS [17] to answer RQ1 and RQ3, (2) the *Winlift* program from the automotive domain [34] to answer RQ2.

All experiments were performed on a Fedora machine with an Intel(R) Xeon(R) CPU E5-1680 v4 @ 3.40 GHz, 128 GB of memory, with a time limit of 1 hour and a memory limit of 70 GB.

A. ERIKA BENCHMARK PROGRAMS

Erika OS [17] is an operating system for IoT devices and is compliant with the OSEK/VDX international standard for operating systems for road vehicles [12]. The benchmark programs range from 36 to 187 lines of code (excluding libraries and header files) with 1 to 3 tasks (2 to 6 threads including the main thread and a thread for alarm). Some of them also include a periodic alarm and/or an ISR (Interrupt Service Routine).

The following two properties from the OSEK/VDX specifications were used:

- S1. Two tasks shall not be runnable at the same time;
- S2. A running task shall terminate at the end.

`S1` is a property that must be satisfied by any embedded software compliant with OSEK/VDX OSs. `S2` is a desirable property, but does not necessarily need to be satisfied, e.g., if a task includes an infinite loop.

1) VERIFICATION USING THE INTERACTION MODEL

We applied our OS-aware interaction model to these programs and used the Spin model checker to verify the two properties specified in LTL as:

$$P1.[]!((task\_state[i] == Running)\&\& \\ (task\_state[j] == Running))$$

$$P2.[]((task\_state[i] == Running) \\ \rightarrow <> (task\_state[i] == Suspended)),$$

where [] and <> represent temporal operators meaning 'always' and 'some time in the future', respectively.

Table 2 shows the result of model checking `S1` and `S2` on the benchmark programs after applying the interaction model. From left to right, each column of each block presents the amount of memory used in MBytes, the verification time in seconds, the search depth, and the numbers of states and transitions traversed during the verification, respectively, for each program. For some cases (test02, test03, and test08), it was not possible to verify the properties within the time and memory limit due to infinitely running periodic alarms. In such cases, the table shows the verification result with a limited number of alarm firings (10 to 20), and the results are written in parentheses.

Spin could not find any violations up to a search depth of 999,999 for S1. Spin also reported that the search depth was deep enough for these programs. For `S2`, Spin's verdict was that 6 out of the 8 programs did not satisfy the property, identifying corner cases as counterexamples. For example, `test02` has a main thread, two tasks (`Task1` with priority 1 and `Task2` with priority 2), and an alarm that activates `Task1` and `Task2` in order. The main thread sets the alarm that periodically activates the two tasks. The counterexample shows a case where `Task1` starts running, but is never able to finish due to the following reason: The alarm fires and activates `Task1` first and `Task2` next. `Task1` first goes into the *Running* state, but is preempted by the higher-priority task `Task2`. It is assumed that `Task1` resumes its execution after the termination of `Task2`, but the alarm fires again, faster than the termination of `Task2`, activating `Task1` and `Task2` once again. As the alarm fires infinitely often, `Task1` never gets a chance to resume and finish its execution. The counterexample shows that the period of time for the firing of the alarm must be longer than the execution time of `Task2`.

We manually analyzed the identified counterexamples to confirm that they were all true alarms.

2) PERFORMANCE COMPARISON WITH CODE-LEVEL MODEL CHECKING

When it comes to the composition of two heterogeneous components, it is not clear whether source-to-model conversion or model-to-source conversion is the better choice for verification performance, especially when an easy-to-use C code model checker such as CBMC [11] exists. We performed the second conversion by translating the OS model into C and composed it with application code. We also annotated

TABLE 2. Verification of S1 and S2 on Erika x86 benchmark programs using the interaction model.

Name	S1					S2					
	Memory(M)	Time(s)	Depth	#States	#Transitions	Memory(M)	Time(s)	Depth	#States	#Trans	Verdict
test01	1,158.36	5.79	801,151	4,802,217	4,804,413	1,194.79	6.1	801,151	4,802,747	4,810,913	T
test02	(34,077.99)	(495)	(1,052)	(158,750,510)	(363,507,880)	(591.939)	(62)	(873)	(2,313,204)	(54,976,054)	F
test03	(128.73)	(0.01)	(142)	(170)	(350)	(128.73)	(0.01)	(142)	(170)	(350)	(T)
test04	14,009.99	148	1,373	60,479,877	106,115,960	(130.78)	(0.11)	(1,043)	(9,620)	(58,713)	F
test05	129.61	0.03	359	4,508	9,962	128.73	0.01	361	149	152	F
test06	1694.30	36.4	107,298	7,916,424	16,560,520	128.83	0.01	233	614	955	F
test07	158.03	0.48	1,415	147,585	327,994	130.39	0.27	1,331	8,280	205,629	F
test08	(1962.13)	(23.4)	(1,967)	(9,250,876)	(20,091,562)	135.08	0.91	724	1,132	754,118	F

TABLE 3. Verification result of S1 and S2 on Erika x86 benchmark programs using CBMC.

Name	S1-6			S1-9			S1-12		
	Time	Mem	Unw.	Time	Mem	Unw.	Time	Mem	Ver.
test01	36	0.38	S						T
test02	419	36.48	I	723	149.23	I	1,076	498.64	I
test03	38	0.57	I	37	0.41	S			T
test04	810	81.51	I	1,456	356.90	I	2,171	1,020.56	I
test05	551	42.56	I	1,487	460.02	I	2,543	2,144.63	I
test06	760	51.32	I	1,263	174.06	I	1,796	492.13	I
test07	1,189	175.91	I	2,501	1,481.13	I	3,937	5,977.40	I
test08	1,432	198.72	I	2,392	976.64	I	3,444	3,081.12	I

synchronization points into the source code, but did not perform C code embedding as this was not necessary.

Table 3 shows the verification performance of CBMC, as the unwind option – the number of loops unwinding – was increased from 6 to 12. For each unwind option, the time, the memory, and whether the unwind option (Unw.) was sufficient (S) or insufficient (I) to ensure sound verification were measured. The number of occurrences of alarms and ISRs was limited to five in these experiments because allowing up to 20 occurrences was too expensive for model checking using CBMC.

We noted that CBMC quickly finished verification of `S1` on test01 and test03 after using unwind options 6 and 9, respectively, without producing any unwind violation. However, it could not determine whether the property was satisfied or not on other programs after running for up to 5,977.40 seconds, as the unwind options used were insufficient. We increased the unwind options to 15, but the result was the same with increased cost. In comparison, code-to-model conversion using the application wrapper and the interaction model finished the same verification within 148 seconds.

We note that it is not feasible to verify (or refute with a reasonable counterexample) these properties without taking the OS behavior into account, as the verification result is not accurate considering the enormous number of false alarms that need to be analyzed and dismissed one by one.

B. THE WINLIFT PROGRAM

Winlift, a control software for opening and closing the windows of a vehicle, is being developed by Metrowerks, Inc., as part of the HC12 OSEKturbo ANSI-C Simulator project [34].[6] The main program comprises 980 lines of code, consisting of 5 tasks, 5 alarms, 1 ISR, 6 external events, and 9 internal events.

The main task, the controller, controls the states of a window, which can be in any of ten states, {UP, DOWN, STOP, CLOSE, OPEN, LOCK, UNLOCK, LOCKED, STALL, REVERSE}. The controller indefinitely waits for external events, computes the output to control the windows, and changes the state of the window. The change of state requires checking external/internal events followed by cancelling and/or setting alarms, setting internal events, and changing output values such as LEDs (Light-Emitting Diodes) and control signals. Other tasks either periodically receive input events, set or cancel alarms, or activate other tasks.

The prototype tool was used to generate a formal OS model and apply the interaction model to compose the OS model with the program, and to check the reachability of each control state of Winlift using Spin. We chose relatively simple reachability checking in this experiment because it is not trivial to see whether each control state is actually reachable, due to the complicated control logic and the lack of documentation.

Our first trial failed to identify a reachable control state after using up 70 GB of memory, due to the state explosion caused by non-deterministic and indefinite occurrences of interrupts and alarms. We were only able to find reachable traces after limiting the number of interrupts and alarms to between 5 to 10.

The left part of Table 4 (labeled with Interaction Model) shows the performance of the reachability checking. We were able to identify execution traces to each control state within 25 seconds using 2 to 4 GBytes of memory. All states except for `UP` were identified as reachable with a maximum of five occurrences of alarms and interrupts. `UP` was not reachable under the same condition, given the memory threshold of 70 GBytes, but required a maximum of eight occurrences of alarms and interrupts, searching over a search depth of 7 million. `STOP` is the initial state and is thus reachable by default. Therefore, we checked whether the control state can go back to `STOP` after changing to other states. We used bitstate hashing with a default hash size of 28 and the weak-fairness option when running the Spin verification in order to speed

[6]It is the largest publicly available program we could find in the automotive domain.

TABLE 4. Reachability checking on Winlift with 5-10 alarm occurrences.

State	Interaction model					Oil-CEGAR [29]			
	Memory(M)	Time(s)	Depth	#States	#Transitions	Time(s)	#Refinements	Analysis time(s)	Total time(s)
CLOSE	2,111	0.04	1,497	6,035	13,052	1,023.505	5	660.75	1684.255
LOCK	2,661	0.04	1,497	6,035	13,052	88.644	0	0.92	89.564
LOCKED	2,844	2.32	2,688	418,259	913,655	695.026	0	0.94	695.966
OPEN	2,111	0.03	1,147	5,144	11,108	709.907	4	661.92	1,371.827
DOWN	2,661	0.05	1,436	6,851	14,732	12,063.295	5	2,183.62	14,246.915
STALL	3,100	13.2	3,551	2,232,514	5,285,618	29,495.099	7	11,997.35	41,492.449
UP	3,838	14.0	7,293,089	2,032,554	2,748,769	18,128.581	7	10,093.52	28,222.101
REVERSE	3,930	24.8	3,994	4,103,457	9,969,870	94,347.096	17	12,775.4	107,122.496
STOP	2,820	2.29	2,474	896,880	2,840	.	.	.	.

up the verification at the cost of increased initial memory consumption.

The right part of Table 4 (labeled with OiL-CEGAR) shows the time required to check the same reachability problem using OS-in-the-loop CounterExample-Guided Abstraction Refinement [29], which is the most efficient approach to date that considers the underlying operating system(s) for the verification of multitasking embedded software. Excluding the interaction model, only the OiL-CEGAR approach was able to identify all reachable states of Winlift without generating false alarms. On the other hand, CBMC [11], Yogar-CBMC [46], and CPA-Checker [2] could not identify any reachable states when code-level verification was performed on the composition of the OS model in C and the Winlift program.

OiL-CEGAR was applied to the composition of the OSEK OS modeled in the input language of the model checker NuSMV [37] and the predicate-abstracted Winlift program [29]. From left to right, this part shows the time for the initial checking, the number of refinement steps necessary to find a trace to the given state, the time needed for false alarm analysis, and the total time spent on reachability checking. We note that 17 refinement steps consuming over 107,122 seconds were required to come up with a true reachable trace for the REVERSE state, while the interaction model used only 3,930 seconds to check the same problem.

Though these two results cannot be compared directly, as they were performed using different modeling languages, they give us a rough idea of the effectiveness of the interaction model.

VII. RELATED WORK

Approaches for verifying multitasking embedded software can be divided into three categories: (1) verification of application programs with a highly abstracted scheduling policy [20], [27], [32], [38], [44], (2) verification approaches for OS [5], [6], [14], [26] that focus on the correctness of either OS models or implementations; and (3) verification of embedded programs with verified OS models [28], [42], [50], [52].

Verification with a highly abstracted scheduling policy has been a main stream in research and practice of model-checking multitasking programs [20], [27], [32], [38], [44], [47]. Works in this category assume arbitrary interleavings among tasks, but reduce verification complexity by either using partial-order reduction, limiting the number of context switches among tasks, or applying CEGAR [10], [47]. These approaches suffer from a high false-alarm rate and/or additional cost for refinements caused by allowing arbitrary sequences of task executions.

The second category includes the Haskell model of seL4 [31], verification methods for OSEK-conformant compilers [14], model checking of the Trampoline OS [6], compositional verification of OS kernels and device drivers [5], the formal OSEK/VDX OS model in the K-framework [52], and modeling and verification of an OS kernel in CSP [26].

A number of approaches realize the importance of addressing OS-related issues and try to compose formal OS models with control software written in C by (explicitly or implicitly) converting C programs into the modeling language used to model the OS [14], [43], [51], [52]. Reference [14] developed verification methods for an OSEK/VDX-conformant code generator that interweaves system calls and application code using a static configuration file. Zhang *et al.* [51] modeled an OSEK/VDX OS in Promela and built a tool for translating application programs written in C into Promela. Other approaches suggest similar ideas using different modeling languages to model the OS kernel, such as Uppaal, Spin, CSP and NuSMV, or translating the application source code into the modeling language [26], [28], [41], [43]. Reference [52] is unique in that the application program is implicitly converted into rewrite logic within the K-framework, which is equipped with language interpreters for C, Java, and JavaScript. However, it suffers from high verification cost due to the use of faithful interpretation of the program code as well as the formal OS model.

Some other approaches have translated concurrent programs into sequential programs considering OS scheduling behavior [44], [50], which can be efficient when the OS has deterministic behavior. This is not realistic as embedded software frequently utilizes alarms and ISRs and it is not clear how these non-deterministic behaviors can be sequentialized without causing any state or transition explosion.

None of the above-mentioned approaches deals with periodic alarms and ISRs.

VIII. DISCUSSION AND CONCLUSION

We have presented an interaction model for the generic composition of OS models with multitasking control programs. The proposed approach provides an OS-aware verification framework for embedded software by utilizing existing verified OS models. Through systematic construction of the interaction model, the error-prone manual conversion can be avoided, the verification of embedded software can become more approachable and effective, and the number of false alarms can be reduced.

A. GENERAL APPLICABILITY

The suggested interaction model can be applied to other modeling languages as long as they provide means for (1) embedding user-defined functions into the model and (2) referring to the OS system state. For example, the modeling language of UPPAAL [21], [23], a well-known model checker suitable for real-time embedded systems, supports specification of user-defined functions, which can be utilized to embed statement blocks. We manually tried to apply the interaction model to the UPPAAL model of the OSEK OS. An inconvenience we found was that UPPAAL's user-defined function does not allow including library functions, which requires more work in automation than using the Spin model. The K-framework [51] may be another candidate. As it is equipped with a C interpreter, building an interaction model within the K-framework would not require more work than using Spin as a modeling language.

The interaction model is generic in a sense that it can also be applied to language-to-language translation without using the C code embedding. As any OS-application composition would require the handling of context switching, a language-to-language translation could adopt the application wrapper by replacing each statement block with a language-specific module.

The soundness of our interaction model is provided in the absence of race conditions, which can be checked without an operating system. Approaches exist that are dedicated specifically to checking race conditions [3], [16], [38].

B. MODEL-BASED VS. CODE-BASED

One might think that code-based verification is more practical and efficient, as we can apply C code model checking directly to the C source code. This is not true if we have to take the OS behavior into account. It is not trivial to apply C code model checking directly to the control program together with the OS source code because the OS implementation typically involves platform-dependent libraries and direct access to the hardware memory space. Abstraction and modeling are necessary.

We can perform abstractions at the code level or translate the OS model into the C language to perform code-level model checking. In this case, however, we lose all the powerful supports of modeling languages, such as implicit support for concurrency, atomicity, non-deterministic choices, and blocking and restarting a process, which must be explicitly implemented in C causing more complexity in verification. Our experiment applying CBMC on the set of Erika benchmark programs shows evidence of this increased complexity and verification cost.

C. SPECIFICS FOR MODEL CHECKING EMBEDDED SOFTWARE

Embedded software depends not only on its underlying operating system, but also on hardware platforms such as memory layout, types of interrupts to handle, and platform-dependent library functions. We manually abstracted them in our experiments and automated only the construction of the interaction model.

Embedded software often includes an infinite loop; e.g., for specifying an idle background task. To avoid waiting for the idle task indefinitely, we had to insert a context switch point within all potentially infinite loops to allow checking the runnability of the task in cases where a higher-priority task preempts the idle task.

D. SCALABILITY

As shown in our experiments, our approach improved scalability compared to that of code-level verification, but could not scale up in the presence of non-deterministic events and periodic alarms. More aggressive but systematic abstractions could be applied to the interaction model to further improve its scalability, such as property-based abstraction and code slicing [8], [9], and compositional verification [4], [5], [22], [36], [49]. We are also working on identifying the minimum occurrences of alarms and ISRs through static dependency analysis among variables, tasks, alarms, and ISRs, to enable sound abstractions of alarms and ISRs.

REFERENCES

[1] R. Alur, "Formal verification of hybrid systems," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 273–278.
[2] D. Beyer and M. Erkan Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 184–190.
[3] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2008, pp. 356–363.
[4] M. G. Bobaru, D. Giannakopoulou, and S. Corina Pasareanu, "Refining interface alphabets for compositional verification," in *Proc. 13th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2007, pp. 292–307.
[5] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, 2016, pp. 431–447.
[6] Y. Choi, "Model checking trampoline OS: A case study on safety analysis for automotive software," *Softw. Test., Verification Rel.*, vol. 24, no. 1, pp. 38–60, Jan. 2014.
[7] Y. Choi, "A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems," *J. Syst. Softw.*, vol. 137, pp. 563–579, Mar. 2018.
[8] Y. Choi, M. Park, T. Byun, and D. Kim, "Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation," *Sci. Comput. Program.*, vol. 103, pp. 51–70, Jun. 2015.
[9] A. Cimatti, M. Roveri, V. Schuppan, and S. Tonetta, "Boolean abstraction for temporal logic satisfiability," in *Proc. Int. Conf. Comput.-Aided Verification*, 2007, pp. 532–546.

[10] E. Clarke, "Counterexample-guided abstraction refinement," in *Proc. 4th Int. Conf. Temporal Logic. Proceedings.*, 2000, pp. 154–169.
[11] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. 10th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2004, pp. 168–176.
[12] OSEK/VDX Consortium. *OSEK/VDX Operating System Specification 2.2.3*. Accessed: Jul. 23, 2020. [Online]. Available: https://ayorho.files.wordpress.com/2011/05/osek_specification.pdf
[13] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Trans. Softw. Eng.*, vol. 38, no. 4, pp. 957–974, Jul. 2012.
[14] H.-P. Deifel, M. Gottlinger, S. Milius, L. Schroder, C. Dietrich, and D. Lohmann, "Automatic verification of application-tailored OSEK kernels," in *Proc. Formal Methods Comput. Aided Des.*, Oct. 2017, pp. 196–203.
[15] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 42–52, Apr. 2009.
[16] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 237–252.
[17] Erika Enterprise. *Erika Realtime Operating System*. Accessed: Jul. 23, 2020. [Online]. Available: http://erika.tuxfamily.org/drupal/
[18] Linux Foundation. *The Zephyr Project*. Accessed: Jul. 23, 2020. [Online]. Available: http://www.zephyrproject.org
[19] *The FreeRTOS Project*. Accessed: Jul. 23, 2020. [Online]. Available: http://www.freertos.org
[20] M. M. Gallardo, C. Joubert, P. Merino, and D. Sanán, "A model-extraction approach to verifying concurrent c programs with CADP," *Sci. Comput. Program.*, vol. 77, no. 3, pp. 375–392, Mar. 2012.
[21] X. Gong, J. Ma, Q. Li, and J. Zhang, "Automatic model building and verification of embedded software with UPPAAL," in *Proc. IEEE 10th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Nov. 2011, pp. 1118–1124.
[22] A. Gupta, K. L. Mcmillan, and Z. Fu, "Automated assumption generation for compositional verification," *Formal Methods Syst. Des.*, vol. 32, no. 3, pp. 285–301, Jun. 2008.
[23] A. Hessel, G. Kim Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using UPPAAL," in *Formal Methods and Testing* Berlin, Germany: Springer, 2008, pp. 77–117.
[24] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
[25] J. Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2003.
[26] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, "Modeling and verifying the code-level OSEK/VDX operating system with CSP," in *Proc. 5th Int. Conf. Theor. Aspects Softw. Eng.*, Aug. 2011, pp. 142–149.
[27] O. Inverso, E. Tomasco, B. Fischer, S. L. Torre, and G. Parlato, "Bounded model checking of multi-threaded programs via lazy sequentialization," in *Proc. Int. Conf. Comput.-Aided Verification*, 2014, pp. 586–602.
[28] D. Kim and Y. Choi, "A two-step approach for pattern-based API-call constraint checking," *Sci. Comput. Program.*, vol. 163, pp. 19–41, Oct. 2018.
[29] D. Kim and Y. Choi, "Model checking embedded control software using OS-in-the-Loop CEGAR," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 565–576.
[30] M. Kim, Y. Kim, and H. Kim, "A comparative study of software model checkers as unit testing tools: An industrial case study," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 146–160, Mar. 2011.
[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal verification of an OS kernel," *Commun. ACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010.
[32] D. Kroening, E. Clarke, and K. Yorav, "Behavioral consistency of C and Verilog programs using bounded model checking," in *Proc. DAC*, 2003, pp. 368–371.
[33] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 613–629, Sep. 2004.
[34] Metrowerks. *Winlift*. Accessed: Jul. 23, 2020. [Online]. Available: https://www.nxp.com/design/software/development-software/codewarrior-development-tools:CW_HOME
[35] M. Musuvathi, Y. W. David Park, A. Chou, R. Dawson Engler, and L. David Dill, "CMC: A pragmatic approach to model checking real code," in *Proc. 5th Symp. Operating Syst. Des. Implement.*, 2002, pp. 75–88.
[36] W. Nam, P. Madhusudan, and R. Alur, "Automatic symbolic compositional verification by learning assumptions," *Formal Methods Syst. Des.*, vol. 32, no. 3, pp. 207–234, May 2008.
[37] *NuSMV: A New Symbolic Model Checking*. Accessed: Jul. 23, 2020. [Online]. Available: http://nusmv.fbk.eu/
[38] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Proc. Int. Conf. Comput.-Aided Verification*, 2005, pp. 82–97.
[39] G. Rodriguez-Navas and J. Proenza, "Using timed automata for modeling distributed systems with clocks: Challenges and solutions," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 857–868, Jun. 2013.
[40] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmäller, "Incremental bounded model checking for embedded software," *Formal Aspects Comput.*, vol. 29, no. 5, pp. 911–931, Sep. 2017.
[41] A. Singh, M. D'Souza, and A. Ebrahim, "Formal verification of datarace in safety critical ARINC653 compliant RTOS," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2018, pp. 1273–1279.
[42] T. Vortler, B. Hockner, P. Hofstedt, and T. Klotz, "Formal verification of software for the contiki operating system considering interrupts," in *Proc. IEEE 18th Int. Symp. Design Diag. Electron. Circuits Syst.*, Apr. 2015, pp. 295–298.
[43] L. Waszniowski and Z. Hanzálek, "Formal verification of multitasking applications based on timed automata model," *Real-Time Syst.*, vol. 38, no. 1, pp. 39–65, Jan. 2008.
[44] X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang, "Data race detection for interrupt-driven programs via bounded model checking," in *Proc. IEEE 7th Int. Conf. Softw. Secur. Rel. Companion*, Jun. 2013, pp. 204–210.
[45] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, pp. 393–423, Nov. 2006.
[46] L. Yin, W. Dong, W. Liu, Y. Li, and J. Wang, "Yogar-CBMC: CBMC with scheduling constraint based abstraction refinement," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2018, pp. 422–426.
[47] L. Yin, W. Dong, W. Liu, and J. Wang, "On scheduling constraint abstraction for multi-threaded program verification," *IEEE Trans. Softw. Eng.*, vol. 46, no. 5, pp. 549–565, May 2020.
[48] A. Zaks and R. Joshi, "Verifying multi-threaded C programs with SPIN," in *Proc. 15th Int. Workshop Model Checking Softw.*, 2008, pp. 325–342.
[49] F. Zaraket, J. Baumgartner, and A. Aziz, "Scalable compositional minimization via static analysis," in *Proc. ICCAD-. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2005, pp. 1060–1070.
[50] H. Zhang, T. Aoki, and Y. Chiba, "Yes! You can use your model checker to verify OSEK/VDX applications," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.
[51] H. Zhang, G. Li, Z. Cheng, and J. Xue, "Verifying OSEK/VDX automotive applications: A spin-based model checking approach," *Softw. Test., Verification Rel.*, vol. 28, no. 3, p. e1662, May 2018.
[52] X. Zhu, M. Zhang, J. Guo, X. Li, H. Zhu, and J. He, "Toward a unified executable formal automobile OS kernel and its applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 1117–1133, Sep. 2019.

YUNJA CHOI received the B.S. and M.S. degrees in mathematics from Yonsei University, Seoul, South Korea, and the M.S. and Ph.D. degrees in computer science from the University of Minnesota, Minneapolis, MN, USA. She worked as a Research Scientist with the Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, before joining Kyungpook National University, in 2006. She is currently a Professor with the School of Computer Science and Engineering, Kyungpook National University, Daegu, South Korea. Her research interests include, but not limited to, software safety analysis, light-weight formal verification, model-based testing, and model-driven component engineering.