Model checking embedded control software using OS-in-the-loop CEGAR

Dongwoo Kim
School of Computer Science and Engineering
Kyungpook National University
Daegu, South Korea
kdw9242@gmail.com

Yunja Choi
School of Computer Science and Engineering
Kyungpook National University
Daegu, South Korea
yuchoi76@knu.ac.kr

***Abstract*—Verification of multitasking embedded software requires taking into account its underlying operating system w.r.t. its scheduling policy and handling of task priorities in order to achieve a higher degree of accuracy. However, such comprehensive verification of multitasking embedded software together with its underlying operating system is very costly and impractical. To reduce the verification cost while achieving the desired accuracy, we propose a variant of CEGAR, named OiL-CEGAR (OS-in-the-Loop Counterexample-Guided Abstraction Refinement), where a composition of a formal OS model and an abstracted application program is used for comprehensive verification and is successively refined using the counterexamples generated from the composition model. The refinement process utilizes the scheduling information in the counterexample, which acts as a mini-OS to check the executability of the counterexample trace on the concrete program. Our experiments using a prototype implementation of OiL-CEGAR show that OiL-CEGAR greatly improves the accuracy and efficiency of property checking in this domain. It automatically removed all false alarms and accomplished property checking within an average of 476 seconds over a set of multitasking programs, whereas model checking using existing approaches over the same set of programs either showed an accuracy of under 11.1% or was unable to finish the verification due to timeout.**

***Index Terms*—CEGAR, embedded OS, multitasking**

I. Introduction

An embedded system consists of a number of devices controlled by software programs. Such control software is typically multitasking and runs on top of an operating system designed for controlling small-scale embedded devices, such as sensors, actuators, brake pedals, engines, etc., which are mostly used in safety-critical domains. The control program, in particular, is tightly coupled with its underlying operating system as they are compiled together to generate a piece of embedded control software, and the behavior of the control program can therefore not be analyzed accurately without taking into account the behavior of the operating system.

However, research and practice in this domain have focused either on the verification of the control program or on the verification of the operating system, independent of each other [1]–[8], due to the huge verification cost when trying to comprehensively verify application software together with the implementation of the operating system.

Verifying embedded control software without considering the underlying OS often results in a high rate of false alarms, as such verification is highly likely to refute a given verification property based on incorrect execution sequences among tasks. For example, a low-priority task may be preempted by higher-priority tasks in a control program. Such behavior is determined by the system configuration and the scheduling policy of the OS used. Most existing approaches that do not consider the underlying OS use sound abstraction of the scheduling behavior such as non-deterministic scheduling of tasks [1], [4]. Our experiments showed that these approaches had an accuracy of only 11.1%, verifying only two of the 18 applications within the time bound.

This work sets out to find an efficient verification method for multitasking embedded control software that reduces false alarm rates as well as verification cost. To this end, we adapted the CEGAR approach [9], [10], where abstraction-verification-refinement iterations are successively performed until a real problem is identified or the given property is verified, to the domain of embedded software by taking the operating system into the loop. Our approach, named OiL-CEGAR (OS-in-the-Loop CEGAR) is unique in that (1) it utilizes models of the operating system, which are assumed to be correct w.r.t. the requirements specifications, and (2) we take advantage of two different types of model checkers, a symbolic model checker, NuSMV [11], for property checking and the C code model checker CBMC [1] for code-based false alarm identification.

For a given property, OiL-CEGAR first tries to verify the property on the composition of the operating system model and the application model abstracted from the program source code. If the property is refuted, the model checker NuSMV generates a counterexample trace including task scheduling information and the control flow of the application program. This task scheduling information is used to construct a mini-OS for the application source code so that CBMC can be used to check the executability of the given trace in the concrete application program. If the trace is executable, then the counterexample is a real alarm. Otherwise, the composition of the OS model and the application model is refined using the trace so that the next iteration of OiL-CEGAR can be performed.

OiL-CEGAR is efficient in identifying real property violations with moderate verification cost, even though more complexity is introduced by taking the OS model into ac-

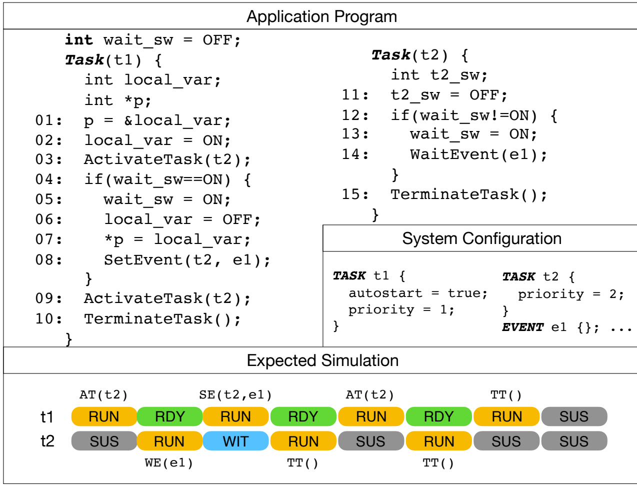

Fig. 1. A multitasking program and its expected execution sequence

count. This is possible through two-way abstractions, code abstraction for the property checking, and OS abstraction for the executability checking, and by utilizing two different model checking approaches suited best for the verification of statemachine-based models and the verification of program source code, respectively.

A prototype implementation of OiL-CEGAR was applied to three sets of experiments. It showed 100% verification accuracy, automatically removed all false alarms, and accomplished property checking within 476 seconds, on average, whereas existing verification approaches using CBMC [1] and Yogar-CBMC [4] showed a verification accuracy of 0% and 11.1%, respectively, over the same set of programs.

The remainder of this paper is organized as follows. After a brief introduction of the research background in Section II, we formally define the basic terminology used throughout this paper in Section III. Section IV introduces the OiL-CEGAR process, Section V explains the abstraction techniques applied to the program source code, and Section VI describes the methods used for constructing models for OiL-CEGAR. An experimental result of OiL-CEGAR is explained in Section VII. We conclude with a brief discussion in Section IX, after summarizing related work in Section VIII.

II. Background

A multitasking application program consists of a set of tasks. 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 application program interacts with its underlying operating system through API functions provided by the OS. Figure 1 shows an example of the embedded application program that will be used throughout this paper. Given the system configuration, the two tasks are expected to be executed

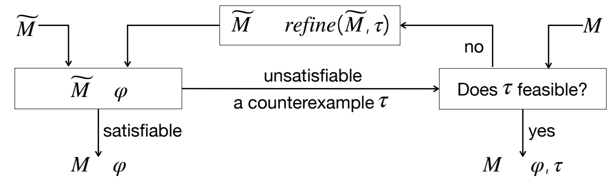

Fig. 2. Counterexample-guided abstraction refinement

as shown in the lower part of the figure; 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 wakes t2. After t2 terminates, t1 activates t2 again, but t2 terminates immediately as *wait_sw* evaluates to ON.

A. Verification of embedded control software

The behavior of an embedded application program is determined by its system configuration, task logic, and OS behavior, which typically produce a unique deterministic trace unless an interrupt occurs. If we abstract the operating system, e.g., by using non-deterministic scheduling, we need to consider multiple execution traces for the same program as context switching may occur at any instruction of each task. For example, there can be up to $15C5 = 3;003$ execution sequences to check for the program in Figure 1 if we assume that a context switch may occur in every line of the program. It is extremely expensive to perform comprehensive verification in this case, both in terms of performing model checking for property checking and in terms of identifying false alarms produced by incorrect task execution sequences.

We note that even software-level dynamic testing of an embedded control program is difficult because embedded software requires a specific hardware platform, platform-specific library functions, and other peripherals such as sensors and event generators. Software-level testing often requires specific simulation environments [12], which are not always available in the development process.

B. Counterexample-guided abstraction refinement

CEGAR [9], [10] is an effective verification method for alleviating verification complexity through successive abstraction-verification-refinement iterations. Figure 2 shows the overall process of CEGAR. It starts with an initial abstraction of a given application $\widetilde{M}$. If the abstract model is verified for a given property φ', then we can safely conclude that the concrete application is verified, as the model is a sound abstraction of the application. Otherwise, the counterexample τ generated from checking φ' is tested on the concrete application to determine whether it is an actual execution trace, as it may be the case that the trace is possible only on the abstract model (false alarm). If it is executable on a concrete application, it shows a property violation of the system (true alarm). Otherwise, τ is used to refine $\widetilde{M}$, which is verified w.r.t. φ' in

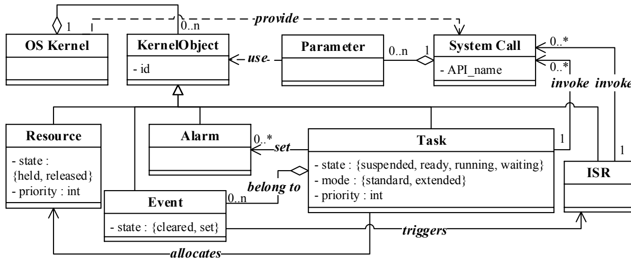

Fig. 3. Structure of an embedded operating system (OSEK/VDX)

the next iteration. This process is repeated until a real alarm is identified or until the property is verified.

CEGAR has been successfully applied in various application domains [4], [13]–[18]. However, it has not been clear whether it can be effectively applied to multitasking embedded software, taking into account the OS behavior.

C. Formal models of embedded operating system

A number of approaches for formally modeling and verifying embedded operating systems have been proposed. Most notably, the construction of seL4 [19] is based on successive refinements of a formal OS model written in Haskell [20], which is thoroughly verified for correctness w.r.t. the functional requirements of a real-time embedded operating system. The other notable example is the formal modeling of embedded operating systems that are compliant with the OSEK/VDX international standard [21]. Approaches include, for example, modeling in Promela [22], CSP [8], NuSMV [23], UPPAAL [24] and the K-framework [25]. OiL-CEGAR is independent of formal models and modeling approaches, but this work adopts the pattern-based automatic model construction approach suggested in [23] to facilitate automation of the OiL-CEGAR process.

Figure 3 shows the internal structure of the embedded operating system used in [23], modeled by referring to the OSEK/VDX international standard. An OS kernel consists of a set of kernel objects, which can be Tasks, Events, Resources, or Alarms. A task (or thread) is the basic building block of embedded software. An embedded OS maintains the internal states of each task which typically consists of {running, ready, waiting, suspended} as in the OSEK/VDX OS, Zephyr [26], FreeRTOS [27], etc., with minor variations. The internal state of each task changes according to the task management and scheduling mechanism in the OS kernel, which is triggered by requests from the application program through API function calls. A task may set a periodic alarm to activate another task or to set an event. An external event triggers a corresponding ISR (Interrupt Service Routine), which may call API functions. The OSEK OS adopts priority-based FIFO scheduling with dynamically changing ceiling priorities for resource allocation in order to avoid the priority-inversion problem.

Reference [23] formally modeled the behavior of each kernel object as a parameterized statemachine (called patterns) and defined an OS model as a synchronized parallel composition of multiple kernel objects whose types and numbers are determined by the system configuration. In addition, an OS generator was defined as a function from configuration vectors to formal OS models. This approach is implemented as a prototype tool for automated model construction by composing the formal patterns specified in the input language of NuSMV [11] or Spin [28]. The generated OS models are validated through property verification using a set of functional requirements identified from the OSEK/VDX international standard. For more details on the OSEK OS and pattern-based OS model construction, please refer to [23].

Our work utilized the prototype tool with a minor extension to the OS model. To enable a focused discussion of OiL-CEGAR, however, throughout this paper an OS model is assumed to be a black-box component where transitions among internal states of tasks are the only externally visible behaviors.

III. Basic definitions

This section defines the terminologies required to explain OiL-CEGAR.

Definition 1. A control flow graph (CFG) for a task $T_{CFG} = (N, E, n_0, n_t)$ is a directed graph where N is a set of statement blocks, $n_0 \in N$ is a unique entry block, $n_t \in N$ is a unique exit block, and $E : N \to N$ is a set of directed control flow edges.

A statement block is a sequence of statements. It is also a unit of atomic execution. Each statement block consists of the maximal number of statements that can be executed in a single control thread. To formally define a statement block, we use the notion of *visible variable* and *visible statement*, similar to the notion introduced in [2].

Definition 2. Visible variables and visible statements

- A visible variable is a variable that is globally accessible or that uses other visible variables. Visible variables include global variables, pointer variables, and variables in shared memory,
- A visible statement is a simple statement that either defines/uses a visible variable or calls an API function.

Definition 3. A statement block is the maximal sequence of statements $a_1, \ldots, a_n$ in a task, where only the first statement in the block is visible and there is no statement that uses any variables defined in previous statements, i.e., $def(a_i) \cap use(a_j) = \emptyset$ for all $i < j \le n$.

Each statement block contains at most one visible statement and is a unit of context switching, i.e., the control flow may be switched to other tasks after the statement block has been executed.

Definition 4. $M = (S, S_0, R)$ is a statemachine, where S is a set of states, S_0 is a set of initial states, and $R \subseteq S \times \Sigma \times G \times S$ is a set of transition relations triggered by a set of events Σ, including a null event, and guarded by a set of guarding conditions G.

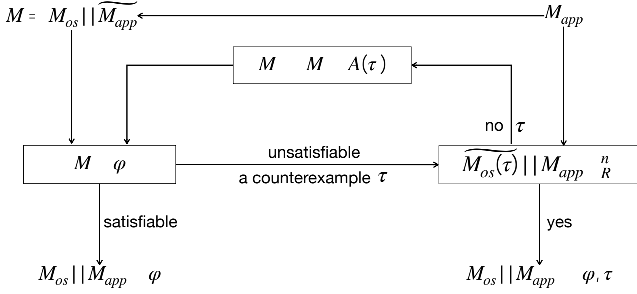

Fig. 4. OS-in-the-Loop CEGAR

A transition $r \in R$ is represented as $s_i \xrightarrow{e[g]} s_j$, where s_i and s_j are the source and target states, respectively. We sometimes abbreviate a sequence of transitions $s_0 \xrightarrow{e_1[g_1]} s_1 \xrightarrow{e_2[g_2]} s_2 \xrightarrow{e_3[g_3]} \cdots \xrightarrow{e_n[g_n]} s_n$ as $s_0 s_1 s_2 \ldots s_n$ to save space.

We use two composition operators on statemachines, $||$ and $\otimes$, which are defined as follows:

Definition 5. $M_1 || M_2 = (S, S_0, R)$ is a synchronous parallel composition of two statemachines $M_1 = (S^1, S^1_0, R^1)$ and $M_2 = (S^2, S^2_0, R^2)$ synchronized over a set of events in Σ, where

- $S = S^1 \times S^2$,
- $S_0 = S^1_0 \times S^2_0$, and
- $R \subseteq R^1 \times R^2$ such that $s^1_i \xrightarrow{e[g]} s^1_j \in R^1$ and $s^2_p \xrightarrow{e'[g']} s^2_q \in R^2$ implies (1) $(s^1_i, s^2_p) \xrightarrow{e[g \wedge g']} (s^1_j, s^2_q) \in R$ if $e = e'$, (2) $(s^1_i, s^2_p) \xrightarrow{e[g \wedge \neg g']} (s^1_j, s^2_p) \in R$, and (3) $(s^1_i, s^2_p) \xrightarrow{e'[\neg g \wedge g']} (s^1_i, s^2_q) \in R$.

A synchronous parallel composition [29] of two statemachines allows each statemachine to perform its own transition while the other stays in the same state.

Definition 6. $M_1 \otimes M_2 = (S, S_0, R)$ is a trace composition of two statemachines $M_1 = (S^1, S^1_0, R^1)$ and $M_2 = (S^2, S^2_0, R^2)$ w.r.t. the set of error states S^2_E in M_2 and a labeling function L over $S^1 \cup S^2$, where

- $S \subseteq S^1$,
- $S_0 \subseteq S^1_0$, and
- $R \subseteq R^1$, where $r : s^1_i \xrightarrow{e[g]} s^1_j \in R$ iff $r \in R^1$ and $\exists r' : s^2_p \xrightarrow{e[g']} s^2_q \in R^2$ such that $s^2_q \notin S^2_E$, $L(s^1_i) = L(s^2_p)$, $L(s^1_j) = L(s^2_q)$, and $g \Longrightarrow g'$.

A trace composition only allows a transition of a statemachine when the other statemachine has an equivalent transition w.r.t. the state labels and transition conditions that does not lead to an error state.

IV. OiL-CEGAR

OiL-CEGAR takes advantage of both CEGAR and the use of the OS in the model checking process with a two-way reciprocal abstraction scheme. It is a variation of CEGAR in that (1) a verified OS model is used to enhance the accuracy of the property checking and (2) the scheduling information in the counterexample generated from the property checking is used to construct a mini-OS, instead of the full-scale OS implementation, for checking the executability of the counterexample on the program source code.

This section provides an intuitive introduction to OiL-CEGAR. Technical details will be provided in later sections.

A. Overview of OiL-CEGAR

Figure 4 illustrates the OiL-CEGAR process. It starts with the composition of a verified OS model M_{os} and an initial sound abstraction $\tilde{M}_{app}$ of a given application program M_{app}. For a given property φ, if the composition model $M = M_{os} || \tilde{M}_{app}$ respects the property, i.e., $M \models \varphi$, then we conclude that the concrete embedded software $OS || M_{app}$ also respects the property. Otherwise, a counterexample τ showing a property violation in the form of a sequence of states in M is generated. As τ can be a false alarm, due to the use of the abstract model $\tilde{M}_{app}$ in the verification, so we need to check its executability on the concrete system $OS || M_{app}$. However, as it is difficult to construct an execution environment for the actual implementation of the OS and the application program, we instead check the reachability of the composition, $\hat{M}_{os}(\tau) || M_{app}$, where $\hat{M}_{os}(\tau)$ is a mini-OS constructed from τ, so that if it terminates in a final state, we can conclude that τ is an executable execution sequence in the concrete program, i.e., a true alarm. Otherwise, a trace τ', a subsequence of τ from the initial state to the first unreachable state of τ, is generated and is used to refine M, which is used in the next iteration of OiL-CEGAR. This process is repeated until either an actual property violation is found or the property is proved.

Two well-known model checkers are used in OiL-CEGAR: the symbolic model checker NuSMV [11] for LTL property checking of $M_{os} || \tilde{M}_{app} \models \varphi$ and the SAT-based C code model checker CBMC [1] for checking the reachability of $\hat{M}_{os}(\tau) || M_{app}$.

B. Construction of $\tilde{M}_{app}$

An application program is a parallel composition of multiple tasks, i.e., $M_{app} = P_1 || P_2 || \ldots || P_n$. We construct an abstraction of M_{app}, $\tilde{M}_{app}$ by abstracting each task, i.e., $\tilde{M}_{app} = M_{P_1} || M_{P_2} || \ldots || M_{P_n}$, where each M_{P_i} is a statemachine representation of each task P_i.

Figure 5 illustrates how each task P_i is modeled as a statemachine using the example code shown in Figure 1. We first construct a control flow graph (CFG) of each task, abstract the CFG by eliminating statements involving $\mathsf{visible}$ variables, and then transform the abstract CFG into a statemachine model by adding transitions from the exit node to the entry node (as a task can be reactivated after its termination) and guarding conditions to each transition to check the internal state of the task. Details of the conversion process will be explained in Section VI.

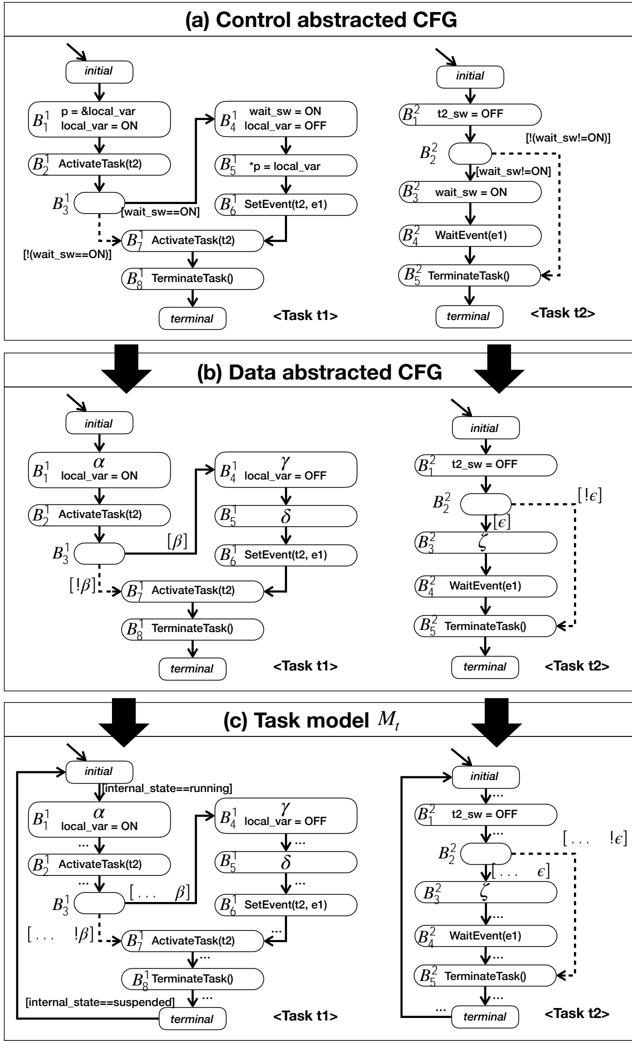

Fig. 5. Construction of the task model

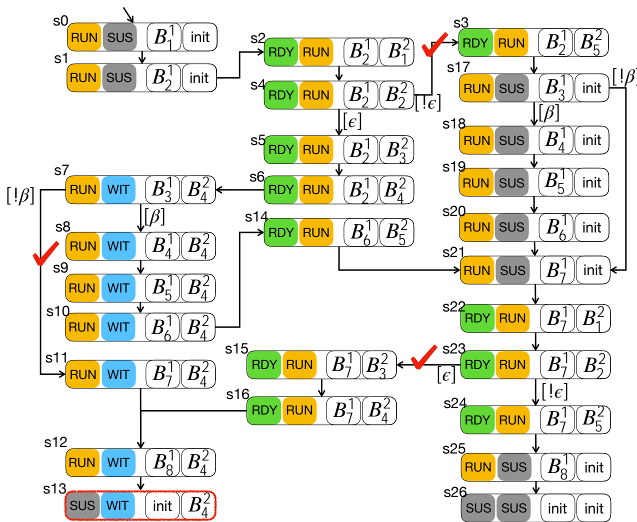

Fig. 6. Composition of the OS model and the application model of the program in Figure 1

$\hat{M}_{app}$ is composed with the OS model. Figure 6[1] illustrates a simplified $M_{os}||\hat{M}_{app}$ for the example in Figure 1, where each state consists of the internal state of task t1, the internal state of task t2, the block number of task t1, and the block number of task t2, respectively. The transition finishes either at $s13$, indicating that task t2 is in the waiting state, or at $s26$, indicating that all the tasks have terminated.

Let us assume that we are checking the following property:

P1. A call to *WaitEvent* shall be followed by a matching call to *SetEvent*.

Model checking this property would generate a counterexample, say $s_0 s_1 s_2 s_4 s_5 s_6 s_7 s_{11} s_{12} s_{13} s_{13} s_{13} \ldots$, where $t2$ would go to the waiting state and stay there forever. is a finite maximal subsequence of a counterexample trace that does not end with a cycle.

C. Executability checking using

Whether the counterexample is a realistic execution trace or not is tested in the concrete application program M_{app}. As the counterexample trace contains information on task execution sequences, we use this information to guide the execution sequence of the tasks in the actual program code.

The task execution sequence is a sequence of states in projected to the states of the OS. For example, $|_{os}$ = $s_0|_{os} s_1|_{os} \ldots s_{12}|_{os} s_{13}|_{os}$ = (RUN,SUS)(RUN,SUS)(RDY,RUN)(RDY,RUN)(RDY,RUN)(RDY,RUN)(RUN,WIT)(RUN,WIT)(RUN,WIT)(SUS,WIT) and its corresponding task execution sequence is $t_1 t_1 t_2 t_2 t_2 t_2 t_1 t_1 t_1,$ where the unit of an execution is a block in the control flow graph as shown in Figure 5 (a). In other words, $B^1_1 B^1_2 B^2_1 B^2_2 B^2_3 B^2_4 B^1_3 B^1_7 B^1_8,$ is the actual sequence of blocks to be executed according to this counterexample trace. However, it is impossible to execute B^1_7 after B^1_3; i.e., $s_7 \rightarrow s_{11}$ is a non-executable transition in because if `wait_sw==ON`, it must execute B^1_4. We use the C code model checker CBMC to identify the first non-executable transition in the counterexample trace using the information on the task execution sequence extracted from the trace called mini-OS, which acts like a test driver.

If the counterexample trace has an unreachable transition in the actual application program, we use the same trace up to the first unreachable state, named $'$, to refine $M_{os}||\hat{M}_{app}$.

D. Refinements using $'$

Once $'$ is identified, we refine $(M_{os}||\hat{M}_{app})$ so that $'$ can be eliminated from its possible execution traces. To this end, we first construct a statemachine $A(') = (S, s_0, T, s_E)$, where $S = \{s \in '.S\} \cup \{s_U\}$ is the set of states in $'$ plus one additional special state s_U representing the universal state; s_0 is the initial state in $'$; s_E is the final state of $'$ representing the error state; and $T = '.T \cup \{s \rightarrow s_U \mid s \neq s_E\} \cup \{s_U \rightarrow s_U\} \cup \{s_E \rightarrow s_E\}$ is a set of transitions in $'$ plus new transitions from each non-error state to the universal state plus self-transitions of s_U and s_E. Figure 7 illustrates $A(')$ for our example for each iteration of OiL-CEGAR.

[1] *RUN*, *RDY*, *WIT*, and *SUS* are abbreviations for *Running*, *Ready*, *Waiting*, and *Suspended*

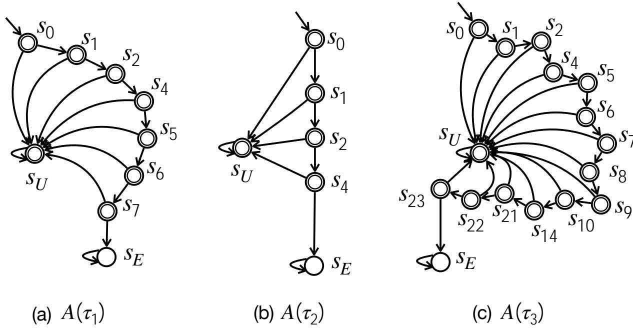

Fig. 7. Refinement bases

$\hat{M}_{app}$ is refined by constructing $(M_{os}||\hat{M}_{app})$ $A(\ ')$, where represents the trace composition of the two statemachines. The trace composition A B allows transitions of A only when A and B share the equivalent source/target states and satisfy both transition conditions, and when the target state is not the error state. For example, the trace composition of $(M_{os}||\hat{M}_{app})$ shown in Figure 6 and $A(\ _1)$ in Figure 7 (a) results in removing the trace $s_0 s_1 s_2 s_4 s_5 s_6 s_7 s_{11}$ from $(M_{os}||\hat{M}_{app})$. The OiL-CEGAR process is repeated from the property checking of $(M_{os}||\hat{M}_{app})$ $A(\ _1)$, producing refinement bases as shown in Figure 7 (b) and Figure 7 (c), and thus, removing traces $s_0 s_1 s_2 s_4 s_3$ and $s_0 s_1 s_2 s_4 \ldots s_{10} s_{14} s_{21} s_{22} s_{23} s_{15}$ in order, before it concludes that the property is satisfied.

V. Abstraction

OiL-CEGAR starts by constructing an application model $\hat{M}_{app}$ as a synchronous parallel composition of statemachines, each representing a task in the application program. This section explains two major abstractions applied to the source code of each task before the application model was composed with the operating system model.

A. Pre-processing

Due to the limited expressive power allowed in the input language of NuSMV compared to that of the implementation language C, we performed two types of pre-processing on the source code. First, user-defined functions were replaced with inline functions, assuming that the program did not contain any infinite recursive calls. Second, composite structures such as arrays and user-defined structures were flattened so that all variables were typed as primitives, assuming that there were no dynamic allocations and that they were thus of fixed size. These assumptions are valid for embedded systems in critical domains, as recommended by the MISRA-C international standard [30].

B. Control abstraction

Multitask programs have multiple threads of control that interleave with each other. This context switching may happen as low as at the instruction level, but it is infeasible as well as uninteresting to consider all possible context switching behaviors in multitasking programs. In this work, we applied control abstraction to the CFG of each task based on the notion of partial-order reduction [31] by defining the statement block as a maximal sequence of statements to be executed without interrupts. The statement block in Definition 3 is a unit of such a context switch, whose execution is independent of concurrently executed transitions of other tasks.

The CFG of each task in the application source code was constructed using the statement block as a unit of execution. The rationale behind this was: (1) Non-visible variables would not be affected by context switching, but visible variables may be, and (2) a call to an API function certainly causes an intervention of the operating system that may require context switching among tasks. In addition, we refined the statement block to separate statements with def-use relations into different blocks, due to the execution semantics of NuSMV, which executes all statements in a block at the same time.

Figure 5 (a) shows the control-abstracted CFG of Figure 1. Blocks B_1^1 and B_4^1 in Figure 5 (a) are the abstraction of lines 01-02 and lines 05-06 of Figure 1, respectively. Line 07 is not blocked together with lines 05-06 because it is a visible statement and also references the variable $local_var$, which is defined in line 06.

C. Data abstraction

A series of selective predicate abstractions was performed on the CFG of the pre-processed, control-abstracted application source code. These abstractions are major sources of non-executable counterexample generation, but greatly help to reduce the complexity of property checking.

1) Abstraction of visible statements: Each visible statement is replaced by an empty statement with a unique symbol.
2) Abstraction of platform-dependent functions: Platform-dependent, user-defined library functions are replaced with stubs that return arbitrary values of the same return type. This abstraction is performed manually, once for each OS platform.
3) Abstraction of branch conditions: If a branch condition references a visible variable, the condition is replaced with a Boolean variable whose value can be non-deterministically assigned during property checking.
4) Numeric data abstraction: Statements involving floating points variables are replaced with empty statements. This is due to a limitation of the model checker NuSMV, which cannot handle floating points.

Compared to typical predicate abstraction [17], the suggested data abstraction is selective, as it is not applied to local variables with fixed size. This is to avoid unnecessary over-approximation, as embedded software frequently uses variables with fixed size due to the stringent memory space.

Figure 5 (b) shows the data-abstracted CFG. The first statement of B_1^1, B_4^1, the statement of B_3^1, B_5^1, B_2^2, and B_3^2 are replaced with empty statements because they reference either a pointer variable p or a global variable *wait_sw*. The second statement of B_1^1 and the second statement of B_4^1 are

not abstracted because *local_var* is a local integer variable that does not have any dependency on any visible variables.

VI. Model construction and implementation

OiL-CEGAR requires the construction of three types of models: the task model constructed from each task CFG after a series of abstractions; the composition of the application model and the OS model; and a mini-OS constructed from a counterexample trace generated from property checking. This section explains the construction of a task model and a mini-OS in more detail.

A. Task model construction

The abstracted CFG for each task is converted into a statemachine $A_t = (S^t, R^t, S^t_0)$ by mapping the set of statement blocks and the set of control flow edges in the CFG to a set of states S^t and a set of transitions R^t, respectively. An additional transition from the final block to the entry block is added to allow the same task to be activated multiple times, removing the final state from the machine. Each transition in the statemachine is guarded by *internal_state[t]==running* to enable the transition only when the task is in the running state. Note that the information about the internal state of each task is maintained by the OS model. Figure 5 (c) illustrates the task models converted from Figure 5 (b), where the common guarding condition *internal_state[t]==running* is omitted from each transition to save space.

The application model $\hat{M}_{app} = A_{t_1} || A_{t_2} || \ldots || A_{t_n}$ is a synchronous parallel composition of task statemachines. Given the number of tasks n in an application program, a parallel composition of an OS model and an application program includes n statemachines for representing tasks internally maintained by the OS and n statemachines, each representing the application logic of each task, synchronized over the set of API function calls and the internal states of the tasks, i.e.,

$$M_{os} || \hat{M}_{app} = \hat{M} || M_{t_1} || \ldots || M_{t_n} || A_{t_1} || \ldots || A_{t_n},$$

where $\hat{M}$ is a parallel composition of kernel objects other than tasks, such as events, resources, alarms, or schedulers. We use $(M_{os} || \hat{M}_{app})|_T = M_{t_1} || \ldots || M_{t_n} || A_{t_1} || \ldots || A_{t_n}$ to represent the projection of $M_{os} || \hat{M}_{app}$ on the task statemachines of the system, as only the task part of the model is of interest for us in the refinement process of the OiL-CEGAR.

B. Mini-OS construction

Given a property φ, the property checking $M_{os} || \hat{M}_{app} \models \varphi$ generates a counterexample if φ is not satisfied by $M_{os} || \hat{M}_{app}$. When $(M_{os} || \hat{M}_{app})|_T$ is a composition of n OS statemachines and n task application statemachines, the counterexample trace σ projected to $(M_{os} || \hat{M}_{app})|_T$ is a sequence of composite states with $2 \times n$ elements. The counterexample trace could be an infinite sequence, but we consider only a finite subsequence of it.

Definition 7. A counterexample trace $\sigma|_T = s_0 s_1 s_2 \ldots s_m$ is a sequence of states with length $m+1$. Each state s_i is a composite state of $2 \times n$ elements $< o^1_i, o^2_i, \ldots, o^n_i, t^1_i, t^2_i, \ldots, t^n_i >$, where n is the number of tasks in the application program.

Model checking may generate an infinite counterexample trace in case it ends with a cycle. In such a case, $\sigma|_T$ is the maximal subsequence of the counterexample trace after removing the final cycle[2].

For simplicity of notation, we will use σ and $\sigma|_T$ interchangeably in the remaining discussions. We will also use $s_i|_{os}$ and $s_i|_{app}$ to represent the OS part and the application part of the composite state s_i in $\sigma|_T$, respectively. The counterexample trace includes information on task execution sequences, i.e., $\sigma|_{os} = s_0|_{os} s_1|_{os} \ldots s_m|_{os}$, which can act as a mini-OS that drives the executability checking of $\sigma|_{app}$ in the application program code.

Definition 8. A mini-OS $\hat{M}_{os}(\sigma) = (S_\sigma, S_0, R_\sigma, s_f)$ is a statemachine constructed from a counterexample trace $\sigma = s_0 s_1 s_2 \ldots s_m$, where $S_\sigma = \{s_i|_{os} \mid s_i \in \sigma\}$, $S_0 = \{s_0|_{os}\}$, $R_\sigma = \{s_i|_{os} \to s_{i+1}|_{os} \mid i = 0 \ldots m-1\}$, and $s_f = s_m|_{os}$.

The mini-OS is just a sequence of finite state transitions where each state indicates the internal states of each task. Using this mini-OS as a test driver, we checked whether the application program is executable until the mini-OS reaches the final state under the following execution semantics of the application program.

$$Exec(i) = \frac{(B_{i_1}, B_{i_2}, \ldots, B_{i_n}), B_{i_j} \to B_{i^0_j}, \hat{M}_{os}(\sigma)|_{i,RUN} = i_k}{(B_{i_1}, B_{i_2}, \ldots, B_{i^0_k}, B_{i_{k+1}}, \ldots, B_{i_n})}$$

When the application program consists of n tasks, the i_{th} execution block is a statement block that belongs to the task assigned to run by the i_{th} state of the mini-OS. With this execution semantics, we applied CBMC model checking to the application program to check whether it finishes all $m+1$ executions as indicated in the mini-OS. If so, the counterexample σ is an executable counterexample, i.e., a true alarm. Otherwise, we can identify the first non-executable transition $s_k|_{os} \to s_{k+1}|_{os}$, where $s_j|_{os} \to s_{j+1}|_{os}$ is executable for all $0 \le j < k$. We used the sequence of states from s_0 to s_{k+1}, named σ^0, to refine $M_{os} || \hat{M}_{app}$.

C. Refinement model construction

σ^0 is an execution trace in $M_{os} || \hat{M}_{app}$ that leads to the first non-executable transition in M_{app}. The goal of refinement is to remove such non-executable execution traces from the model until a counterexample trace executable in the application program is found or until no more counterexample traces exist.

We first constructed a statemachine $A(\sigma^0)$ that included all states and transitions in σ^0 plus an additional state s_U to represent the *Universal* state and transitions from each non-error state to s_U.

[2]NuSMV provides a counterexample trace indicating the start of a cycle, if there is any.

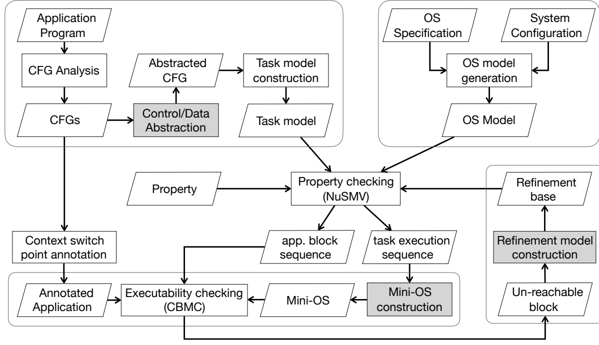

Fig. 8. Overall process of a prototype tool for OiL-CEGAR

Definition 9. A refinement base $A(\sigma') = (S', s_0', R', s_E')$ is a statemachine constructed from a non-executable counterexample trace $\sigma' = s_0 s_1 s_2 \ldots s_k s_{k+1}$, where

- $S' = \{s_i \mid s_i \in \sigma'\} \cup \{s_U\}$,
- $s_0' = s_0$,
- $s_E' = s_{k+1}$,
- $R' = \{s_i \xrightarrow{e[g]} s_{i+1} \mid i = 0 \ldots k\} \cup \{s_i \xrightarrow{e'[g']} s_U \mid i \leq k, (e' \neq e \wedge g' = true) \vee g = \neg g'\} \cup \{s_{k+1} \rightarrow s_{k+1}\} \cup \{s_U \rightarrow s_U\}$.

We used the refinement base to trace-compose, as defined in Definition 6, with the verification model using the labeling function L. Here, L is defined over $S \cup S'$ as $L(s) = L(s')$ iff $s = s'$ or $s = s_U$ or $s' = s_U$. Since there is a transition from any non-error state to s_U in $A(\sigma')$ and s_U has the same label as any other state, $M \| A(\sigma')$ allows all transitions in M except for the sequence of transitions from the initial state to the error state in σ'.

D. Implementation

The prototype tool we used to implement OiL-CEGAR consisted of three components for model generation, executability checking, and refinement. Figure 8 illustrates the overall process of the prototype tool, where the implementation for representing statemachine models in NuSMV follows the approach described in [32]. The formal OS model used in our tool is from the automotive domain and is based on the OSEK/VDX international standard [21], [23].

The construction of $\hat{M}_{os}(\tau) \| M_{app}$ for executability checking using CBMC required some instrumentation: (1) encoding $\hat{M}_{os}(\tau)$ in C as a test driver, and (2) inserting assertions for checking reachability. The `main` function in the right part of Figure 9 shows the test driver defined from $\tau|_{os}$, where *assert(0)* is inserted after each call to the runnable task. The corresponding execution sequence of statement blocks is encoded in an array as indicated in $\tau|_{app}$. Each statement block in the application source code is annotated with a unique label and a program counter. Whenever the task is called by the `main`, it jumps to the corresponding statement block which is annotated with a unique label. An assumption, *assume(c)*, is also annotated before each statement block, which rules out program paths that do not respect the condition *c* during CBMC model checking. This assumption is used to check only those execution paths that follow $\tau|_{app}$ and to ignore all other execution sequences of statement blocks. Each block ends with saving the label of the next statement block to be executed in *pc* and returning control to `main`.

Annotated Application	$\tau\|_{app}$	$\tau\|_{os}$

Task(t1) {
static int local_var;
goto_pc_t1();
pc_t1_1:
assume(B_t1_1==B[b_idx++]);
p = &local_var;
local_var = ON;
pc[t1] = 2; return;
pc_t1_2:
assume(B_t1_2==B[b_idx++]);
ActivateTask(t2);
pc[t1] = 3; return;
...
	int pc[2]={1,1};
	int b_idx=0;
	Block B[]={
	B_t1_1,
	B_t1_2,
	B_t2_1,
	B_t2_2,
	B_t2_3,
	B_t2_4,
	B_t1_3,
	B_t1_7,
	B_t1_8,
	};
		int main() {
		Task(t1); assert(0);
		Task(t1); assert(0);
		Task(t2); assert(0);
		Task(t2); assert(0);
		Task(t2); assert(0);
		Task(t2); assert(0);
		Task(t1); assert(0);
		Task(t1); assert(0);
		Task(t1); assert(0);
		}

Fig. 9. Annotated task and C-encoded mini-OS

If the task sequence is executable, CBMC is supposed to find verification failures for all assertions specified in `main`. Otherwise, we can identify the first assertion that did not fail, meaning that the task (and its corresponding statement block) called before the assertion checking is non-executable.

VII. Experiments

We conducted three sets of experiments to evaluate the effectiveness of OiL-CEGAR using a formal OS model based on the OSEK/VDX international standard for automotive OSs [23] and using the following three test sets:

TS1. A set of application programs running on Lego Mindstorms NXT [33],

TS2. a set of test programs from a commercial conformance test suite used for certifying implementations of OSEK/VDX-based OSs [34], and

TS3. a window controller program from the automotive domain [35].

We chose the programs in TS1 for the sake of fair comparison because they are open to the public, even though they are quite small in scale. TS2 is a more realistic test set as it comes from domain experts, but it is not open to the public. The experiments with TS3 were done to check the feasibility of applying OiL-CEGAR in a program with high complexity.

All experiments were performed on a Fedora Linux-based machine with a 3.4-GHz Intel Xeon E5-1680 CPU and 128 GB of memory. We used CBMC version 5.10 for executability checking, and NuSMV version 2.6.0 with dynamic variable reordering and cone-of-influence reduction. These options were used to accelerate the verification performance.

A. Effectiveness of OiL-CEGAR

The first experiment was aimed at evaluating the effectiveness of OiL-CEGAR using TS1 and TS2. TS1 contained eight programs with varying sizes from 35 to 87 lines of codes and two to three tasks in each application. TS2 contained 14 test programs with 172 to 571 lines of code and five to seven

TABLE I
PROPERTY CHECKING RESULTS ON TS1 AND TS2

App	#T	LoC	B(NB)	Verification Time(s)	Verification #R	Exec. chk. Time(s)	Result	Verif. w/o OS - CBMC Time(s)	Verif. w/o OS - CBMC #U	Verif. w/o OS - CBMC Result	Verif. w/o OS - YOGAR-CBMC Time(s)	Verif. w/o OS - YOGAR-CBMC #U	Verif. w/o OS - YOGAR-CBMC Result
btmaster	3	87	5(1)	5.415	-	0.62	violated	0.38	2	WT	2.96	2	WP
btslave	3	86	5(1)	5.330	-	0.61	violated	0.37	2	WCS	2.12	2	WP
cubbyhole	3	37	2(0)	0.616	-	-	satisfied	1.59	2	WP	2.85	6	WP
eds	3	60	4(0)	0.656	-	-	satisfied	0.44	2	WP	42.37	6	WP
eventtest	2	35	6(4)	0.258	-	-	satisfied	614.05	12	WP	0.39	10	satisfied
message	3	68	5(2)	4.076	-	-	satisfied	0.39	2	WCS	0.30	6	WCS
resourcetest	2	54	9(0)	0.475	-	-	satisfied	-	6	TO	-	5	TO
tttest	2	48	6(4)	1.133	-	-	satisfied	-	12	TO	0.23	5	satisfied
confo_1	6	172	4(2)	40	0	0.131	violated	0.47	2	WCS	0.55	2	WP
confo_2	5	329	9(0)	735	4	0.145	satisfied	2.41	2	WP	0.57	2	WP
confo_3	6	409	9(0)	498	1	0.350	satisfied	2.69	2	WCS	0.69	2	WP
confo_4	5	419	6(0)	360	3	0.158	violated	1.05	2	WP	1.79	6	WP
confo_5	5	419	13(0)	2166	3	0.224	satisfied	0.94	2	WP	0.53	2	WP
confo_6	6	305	17(0)	638	3	0.168	satisfied	51.29	2	WP	-	2	TO
confo_7	6	505	12(0)	205	1	0.173	satisfied	1.02	2	WCS	0.42	2	WP
confo_8	6	317	17(0)	613	3	0.216	satisfied	58.41	2	WP	-	2	TO
confo_9	5	362	11(0)	1521	3	0.153	satisfied	72.53	2	WCS	122.91	2	WP
confo_10	7	346	15(0)	2904	3	0.277	satisfied	2.65	2	WP	0.64	2	WP
confo_11	6	571	21(0)	280	1	0.118	violated	74.25	2	WT	-	2	TO
confo_12	6	276	5(0)	157	0	0.211	violated	0.35	2	WP	0.31	2	WP
confo_13	6	310	12(0)	169	1	0.229	satisfied	0.58	2	WP	0.32	2	WP
confo_14	6	308	11(0)	167	1	0.236	satisfied	0.88	2	WP	0.42	2	WP

tasks. These programs were verified with the same property P1, "A call to *WaitEvent* shall be followed by a matching call to *SetEvent*".

Column one to eight of Table I show the verification results using OiL-CEGAR. From left to right, the columns show the application id, the number of tasks, the lines of codes, the number of branch statements with the number of nested branch statements (in parentheses), the time needed for property checking, the number of required refinement iterations, the time for executability checking of the generated counterexamples, and the final verdict of OiL-CEGAR, respectively.

Applications on TS1 reported two counterexamples, which turned out to be true alarms, among the eight properties checked using NuSMV. Six of them were verified to be true in the first round of OiL-CEGAR. All these applications were checked without further refinements as they did not contain many branch conditions that were subject to abstractions. Among the programs verified to be true, *message* took far longer to check its properties, mainly because it has more tasks containing several nested loops. While verifying the 14 applications in TS2, a total of 41 property checks using NuSMV were performed (including verification after refinements); 31 of them reported counterexamples and ten were verified as having no property violations. Executability checking was performed on these 31 counterexamples and 27 (87.09%) of them were found to be non-executable; these were used for model refinement. On average, 1.92 refinements were required on those 14 applications, which ultimately verified ten programs and refuted four.

We manually analyzed all 22 programs to ensure that our verification results using OiL-CEGAR are accurate.

B. Comparison with CBMC and Yogar-CBMC

In the second experiment, we applied two representative C code model checking tools (CBMC [1] and Yogar-CBMC [4]) to TS1 and TS2 to compare the verification accuracy of OiL-CEGAR to that of the best-known existing verification approaches for multitask programs. CBMC is the most stable and widely used C code model checker for the verification of multi-threaded programs under arbitrary interleavings among threads. Yogar-CBMC implements CEGAR in CBMC and is the 2017, 2018, and 2019 winner of the SV-COMP competition in the verification of concurrent programs.

We successively increased the loop bound from 2 to 21 when applying CBMC, until it either found a counterexample or verified the property without violating the unwinding assertion. Yogar-CBMC automatically sets the loop bound during the CEGAR process. We used a time bound of one hour for both cases.

Columns nine to fourteen of Table I show the verification results. To the right of column nine, the columns show the verification time, the number of loop unwindings, the verification result of CBMC, and the verification result of Yogar-CBMC, respectively.

CBMC and Yogar-CBMC were unable to find any executable counterexample trace for any of the six programs supposed to violate the property. Yogar-CBMC was able to verify two out of sixteen programs correctly, but CBMC was not able to verify any of them. Even though they identified counterexamples relatively quickly, they were all false alarms caused by infeasible context switches (WCS), caused by allowing two instances of a task running at the same time (WT) or by allowing a task with lower priority to run prior to a task with higher priority (WP), or took more than one hour

TABLE II
REACHABILITY CHECKING RESULTS ON WINLIFT

State names	Verification Time(s)	#R	Exec. chk. Time(s)	Result	C.E. len
close	1023.505	5	660.75	violated	17
lock	88.644	0	0.92	violated	14
locked	695.026	0	0.94	violated	19
open	709.907	4	661.92	violated	17
down	12063.295	5	2183.62	violated	22
stall	29495.099	7	11997.35	violated	21
up	18128.581	7	10093.52	violated	22
reverse	94347.096	17	12775.4	violated	23

of time bound (TO). Note that multiple activations of the same task are possible, but only one of the activated tasks should run if priority-based FIFO scheduling is used. As CBMC and Yogar-CBMC do not take this situation into account, false alarms such as WT may occur.

The programs *eventtest*, *resourcetest*, and *tttest* are costly to verify, taking far more time for counterexample generation or going over the time limit. This is because they contain a nested loop or a large number of iterations, e.g., five million iterations in one example.

C. Scalability to complex systems with alarms and ISRs

The last experiment aimed to check how expensive it is to apply OiL-CEGAR to an application program with multiple alarms and interrupt service routines (ISRs). Alarms periodically fire requests to OS services. ISRs handle interrupts, which may occur at any time. They usually have higher priority than tasks, preempting the running task at arbitrary times. This is a major source of complexity in the verification of multitasking programs.

The automotive window controller program in TS3 comprises 980 lines of code, consisting of five tasks, five alarms, and one ISR. The controller has eight system states: close, lock, locked, open, down, stall, up, and reverse. OiL-CEGAR was applied to this program w.r.t. property P2:

P2. (For each system state) The system state is not reachable from the initial state.

Table II shows the verification results. OiL-CEGAR successfully identified counterexamples for all eight states, showing that all states were reachable after 0 to 17 refinement iterations. The length of the generated counterexamples ranged from 14 to 23. The cost for the verification was quite high. For example, checking *reverse* took over 26 hours for 17 iterations of property checking and over 3.5 hours for executability checking using CBMC. However, the verification result was 100% accurate.

D. Threats to validity

OiL-CEGAR guarantees that property violations are found when a sound OS model is used, but does not guarantee that all false alarms will be identified, unless the model is also complete. The use of a sound OS model is a necessary prerequisite for OiL-CEGAR, as it may not be able to identify real alarms in the first place if this is not the case, which is a common requirement for most formal model-based approaches. Available formal OS models [5]–[8] are sound, but might not be complete w.r.t time-dependent behaviors due to the use of relative timing. Our experiments showed 100% verification accuracy because the properties were not time-dependent.

The pre-processing applied to the application source code, such as the flattening of data structures and the use of fixed-size primitive data types, assumes that the program does not contain any data structure with undefined size, as recommended by the MISRA-C standard. If this assumption is not fulfilled, the pre-processing might not be sound and should better be replaced by data abstraction [36] or predicate abstraction [17], which guarantee soundness. We also limited the value ranges of the fixed-size variables in the model because NuSMV is not good at dealing with variables over a large range. This data abstraction should be performed with care, e.g., by using value analysis [37], in order not to undermine the soundness of the model.

Though our experiments showed great improvement of verification accuracy, OiL-CEGAR might not be able to identify non-executable counterexample traces, as we considered only finite subtraces of possible infinite traces. Similar to existing CEGAR approaches, OiL-CEGAR might not terminate as it might refine the model infinitely, e.g., when a task contains an infinite loop.

Our experiments were limited to API call constraint checking and reachability checking. However, OiL-CEGAR can also be used for general property checking with minor changes in task annotation, as NuSMV is capable of checking LTL and CTL properties and CBMC is used only for a bounded search of the counterexample trace.

VIII. RELATED WORK

Numerous approaches for verifying embedded control software exist, which can be divided into three categories: (1) verification of application programs with a highly abstracted scheduling policy [1]–[4], [38]; (2) verification approaches for OS [5]–[8] that focus on the correctness of either OS models or implementations; and (3) a number of recent works on the verification of embedded programs with verified OS models [25], [32], [39]–[41].

A. Verification with a highly abstracted scheduling policy

This approach has been a main stream in research and practice for model checking multitasking programs. The approaches in [1]–[4], [38] assume arbitrary interleavings among tasks, but reduce verification complexity by either using partial order reduction or limiting the number of context switches among tasks. Lazy-CSeq [3], [42] further improves verification performance by predicting a range of values on numeric typed variables [43]. References [4], [44] apply CEGAR under the assumption of arbitrary interleavings among tasks. Notably, references [4] found that most of the CNF formulas generated by CBMC are related to the scheduling constraint that checks sequential consistency [45] of shared variables, and applied

CEGAR on the scheduling constraint to achieve better performance.

These approaches suffer from a high rate of false alarms by allowing arbitrary sequences of task executions, as demonstrated through our experiments.

B. Verification of application software with OS

There are approaches that use verified OS models to reason about application programs. References [22], [32], [40], [46] verified embedded programs with a formal OS model using Spin [28] or NuSMV by translating the application program into the modeling language used to model the operating system. Due to the difference in expressiveness between the implementation language and the modeling language, it is unavoidable that these approaches introduce abstractions into the translation process, which can be a source of false alarms. It is also predicted that faithful translation would result in high verification cost. These issues are not treated in the existing approaches. Reference [25] 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.

C. Refinement methods

Techniques for improving the efficiency of model refinements have been an active research issue [4], [13]–[16]. Studies [4], [13], [14] refine the CNF formula using SAT-core [47], which is a subset of the formula that is sufficient for establishing inconsistency with a counterexample. In trace abstraction refinement [15], [16], the program is abstracted so that arbitrary execution of statements is possible and non-executable counterexamples are iteratively excluded from the model by checking the post-condition of each statement. The refinement in OiL-CEGAR is conceptually similar to trace abstraction refinement, but is different in that it uses the scheduling information of the counterexample and the model checker CBMC to identify non-executable counterexamples.

IX. Discussion

We have presented a novel approach named OiL-CEGAR for accurately verifying multitasking embedded software using a formally verified OS model in the CEGAR process. OiL-CEGAR also alleviates the burden of executability checking in the domain of embedded software, which requires a platform-dependent HW/SW environment setting, by constructing a mini-OS that can act as a test driver simulating feasible execution sequences of multiple tasks. Our experimental results revealed that OiL-CEGAR had 100% verification accuracy with moderately increased cost, while CBMC and Yogar-CBMC showed an accuracy of under 11.1% and all the counterexamples included incorrect task execution sequences.

A. Why model-based?

One might argue that code-based verification is more practical and efficient, as model-based approaches require extra work, whereas we can apply C code model checking directly to C source code. This is not true if we have to take the OS behavior into account because the OS implementation typically involves platform-dependent libraries and direct access to hardware. Abstraction and modeling are therefore necessary.

We can perform abstractions at the code level or translate the OS model into the C language and perform code-level model checking. If we do this, however, we lose all the powerful support of the modeling language, such as implicit support for concurrency, atomicity, and blocking and restarting of a process. These constructs are essential for an OS and must be explicitly modeled in C, resulting in higher complexity in verification. In addition, model-based verification performs property checking over infinite traces while code-based verification is typically limited to finite traces.

In fact, we did try code-level model checking using CBMC and Yogar-CBMC on the same set of programs used in the experiments, by converting the OS model into the C language. Though code-level model checking was faster than OiL-CEGAR on small-scale programs such as TS1 and TS2, neither CBMC nor Yogar-CBMC were able to find any reachable state on TS3 within 26 hours, the longest OiL-CEGAR took to find each reachable control state. CBMC was able to determine that the lock state was reachable within 20 minutes only after we manually removed four tasks from Winlift and limited the number of alarm occurrences to five.

B. Trace refinement vs. predicate refinement

OiL-CEGAR uses trace refinement after identifying false alarms, instead of using typical predicate refinement as in SLAM or BLAST [17], because it is simpler to apply it to NuSMV models using constraints and invariants. The two approaches are orthogonal to each other, one refining predicates considering all execution paths and the other pruning infeasible traces considering all possible values of the predicate for a given trace.

C. Scalability

The use of a formal OS model in the refinement loop increases both verification accuracy and verification cost. The cost for checking complex embedded software with multiple alarms and ISRs is still very high, but, to the best of our knowledge, this is the first experiment showing high verification accuracy in this domain under such complexity. Improving the scalability of OiL-CEGAR is our next research goal.

Acknowledgments

This research has been 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, ICT (No. 2017M3C4A7068175).

REFERENCES

[1] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of c and verilog programs using bounded model checking," in *Proceedings of the 40th Annual Design Automation Conference*, 2003, pp. 368–371.

[2] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *International Conference on Computer Aided Verification*, 2005, pp. 82–97.

[3] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded c programs via lazy sequentialization," in *International Conference on Computer Aided Verification*, 2014, pp. 585–602.

[4] L. Yin, W. Dong, W. Liu, and J. Wang, "Scheduling constraint based abstraction refinement for multi-threaded program verification," *CoRR*, vol. abs/1708.08323, 2017.

[5] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 431–447.

[6] Y. Choi, "Model checking trampoline OS: a case study on safety analysis for automotive software," *Software Testing, Verification and Reliability*, vol. 24, no. 1, pp. 38–60, 2014.

[7] H.-P. Deifel, M. Göttlinger, S. Milius, L. Schröder, C. Dietrich, and D. Löhmann, "Automatic verification of application-tailored OSEK kernels," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 196–203.

[8] 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 *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, 2011, pp. 142–149.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of the 12th International Conference on Computer Aided Verification*, July 2000, pp. 154–169.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 58–70.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," in *International conference on computer aided verification*. Springer, 1999, pp. 495–499.

[12] T. Kuroiwa, Y. Aoyama, and N. Kushiro, "A hybrid testing environment between execution test and model checking for IoT system," in *IEEE International Conference on Consumer Electronics (ICCE)*, 2019, pp. 1–2.

[13] K. L. McMillan, "Lazy abstraction with interpolants," in *International Conference on Computer Aided Verification*, 2006, pp. 123–136.

[14] S.-W. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong, "Interpolation guided compositional verification (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 65–74.

[15] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *International Static Analysis Symposium*, 2009, pp. 69–85.

[16] F. Cassez and F. Ziegler, "Verification of concurrent programs using trace abstraction refinement," in *Logic for Programming, Artificial Intelligence, and Reasoning*, 2015, pp. 233–248.

[17] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast: Applications to software engineering," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, Oct. 2007.

[18] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.

[20] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[21] OSEK Group, "OSEK/VDX operating system specification," 2005.

[22] H. Zhang, G. Li, Z. Cheng, and J. Xue, "Verifying OSEK/VDX automotive applications: A spin-based model checking approach," *Software Testing, Verification and Reliability*, vol. 28, no. 3, p. e1662, 2018.

[23] Y. Choi, "A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems," *Journal of Systems and Software*, vol. 137, pp. 563–579, 2018.

[24] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal—a tool suite for automatic verification of real-time systems," in *International hybrid systems workshop*. Springer, 1995, pp. 232–243.

[25] 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 Transactions on Reliability*, pp. 1–17, 2018.

[26] "Zephyr project," https://www.zephyrproject.org/.

[27] "Freertos - market leading rtos for embedded systems with internet of things extensions," https://www.freertos.org/.

[28] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[29] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[30] *MISRA-C:2012: Guidelines for the use of the C language in critical systems*, 2013.

[31] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," in *International Conference on Computer Aided Verification*. Springer, 1991, pp. 332–342.

[32] D. Kim and Y. Choi, "A two-step approach for pattern-based API-call constraint checking," *Science of Computer Programming*, vol. 163, pp. 19–41, 2018.

[33] "nxtOSEK: ANSI C/C++ with OSEK/μITRON RTOS for LEGO MINDSTORMS NXT," http://lejos-osek.sourceforge.net/.

[34] Y. Choi and T. Byun, "Constraint-based test generation for automotive operating systems," *Software & Systems Modeling*, vol. 16, no. 1, pp. 7–24, 2017.

[35] "WinLift application distributed with CodeWarrior V5.1," https://www.nxp.com/support/developer-resources/software-development-tools/codewarrior-development-tools/downloads:CW_DOWNLOADS.

[36] Y. Kesten and A. Pnueli, "Control and data abstraction: The cornerstones of practical formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 328–342, 2000.

[37] G. Canet, P. Cuoq, and B. Monate, "A value analysis for c programs," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009, pp. 123–124.

[38] X. Wu, Y. Wen, L. Chen, W. Dong, and J. Wang, "Data race detection for interrupt-driven programs via bounded model checking," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, 2013, pp. 204–210.

[39] T. Vörtler, B. Höckner, P. Hofstedt, and T. Klotz, "Formal verification of software for the Contiki operating system considering interrupts," in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2015, pp. 295–298.

[40] H. Zhang, T. Aoki, and Y. Chiba, "Yes! you can use your model checker to verify OSEK/VDX applications," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[41] Y. Chung, D. Kim, and Y. Choi, "Modeling OSEK/VDX OS requirements in C," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2017, pp. 398–407.

[42] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Lazy-CSeq: a lazy sequentialization tool for C," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014, pp. 398–401.

[43] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato, "Concurrent program verification with lazy sequentialization and interval analysis," in *International Conference on Networked Systems*, 2017, pp. 255–271.

[44] C. Popeea and A. Rybalchenko, "Threader: a verifier for multi-threaded programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2013, pp. 633–636.

[45] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[46] L. Waszniowski and Z. Hanzálek, "Formal verification of multitasking applications based on timed automata model," *Real-Time Systems*, vol. 38, no. 1, pp. 39–65, 2008.

[47] A. Belov, H. Chen, A. Mishchenko, and J. Marques-Silva, "Core minimization in SAT-based abstraction," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 1411–1416.