Formal Specifications of Real-Time AUTOSAR-Compliant Operating Systems

Drona Nagarajan
EAS
Graz University of Technology
Graz, Austria
drona.nagarajan@tugraz.at

Tobias Scheipel
EAS
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Marcel Baunach
EAS
Graz University of Technology
Graz, Austria
baunach@tugraz.at

Abstract

The **AUT**omotive **O**pen **S**ystem **AR**chitecture (AUTOSAR) is a set of documents that provide guidance for implementing automotive software. The Software Specifications of the Operating System (SWS_OS) standardizes interfaces and functional requirements for an AUTOSAR-compliant OS. Merely implementing the required interfaces is not sufficient to demonstrate AUTOSAR compliance; it is important to show that the interaction between interfaces meets the functional requirements outlined in the SWS_OS.

This paper introduces a method for formally specifying and verifying that a given implementation meets the SWS_OS specifications. Our approach involves formally defining the functional requirements and using formal reasoning to demonstrate that an implementation adheres to these requirements. We have applied this method to a widely-used real-time operating system (RTOS) to prove its compliance with AUTOSAR by ensuring that it satisfies the necessary functional properties.

CCS Concepts

• **Mathematics of computing** → **Solvers**; • **Computer systems organization** → *Real-time systems*; **Real-time system specification**.

Keywords

AUTOSAR, Dafny, Formal Specifications, Predicate Transformers

ACM Reference Format:
Drona Nagarajan, Tobias Scheipel, and Marcel Baunach. 2024. Formal Specifications of Real-Time AUTOSAR-Compliant Operating Systems. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 06–08, 2024, Porto, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3696355.3699706

Availability of Artifacts

Throughout the paper, details of the Dafny implementation are omitted for the sake of readability or due to space constraints. In such cases, the symbol (§) indicates that details can be found **online**.

RTNS 2024, November 06–08, 2024, Porto, Portugal

ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3699706

1 Introduction

Critical automotive applications depend on the correct implementation of the underlying operating system. The design and development of the OS is guided by the AUTOSAR SWS_OS that standardizes the Application Programming Interfaces (APIs) and the functional behavior behind these APIs for all AUTOSAR-compliant OS.

Usually, these requirements are interpreted by experts, who then implement the OS based on their knowledge and expertise. The implementation is then tested against a set of test cases to show the compliance between the implementation and the specifications in the SWS_OS [1]. However, formal proof that the implementation indeed respects the specifications remains a challenging task.

The SWS_OS is organized into modules, with each module specifying the required APIs and functional requirements. This results in an implementation that contains a large code base, typically written in C, which addresses the functional requirements and implements the required APIs of each module. Moreover, the APIs are interdependent in the sense that a sequence of API calls is involved in satisfying a given functional requirement.

The challenge is not only to verify the correctness of the implementation of the various APIs but also to prove that the interactions between these APIs are consistent with the specifications. In this regard, many approaches are proposed that range from the generation of test cases from a formal model [8, 16] to checking the functional behavior of a model of the OS against the specifications [2, 4]. We propose an approach that uses formal reasoning techniques to verify that the code adheres to the specifications. We first formalize the specifications as formulae in first-order logic. Then, using formal reasoning, we abstract the implementation of the APIs into a first-order logic formula. Finally, we implement these first-order logic formulas in a verification-aware programming language called Dafny [11], to verify compliance.

The main contribution of our approach is to formalize the expected behavior of the OS. By expected behavior, we mean the API call sequence that leads to a state where the given functional requirement is satisfied. The sequence of API calls is a consequence of the specifications of APIs involved in the functional requirement. The expected behavior of the OS is encoded in Dafny as a lemma. Then, using the Dafny theorem prover, we show that the interactions between the abstraction of the various APIs satisfy the required functional requirement.

The API specifications are abstractions of the API derived as a formula in first-order logic using formal reasoning techniques. These specifications are an equivalent representation of the actual implementation of the APIs. Since the abstractions are equivalent to

the C code, the proof that the functional requirements are satisfied directly extends to the C code.

2 Background

An automotive application running on top of an AUTOSAR-compliant OS consists of a set of tasks that implement the intended logic of the application. These tasks have well-defined properties such as periods, priorities, shared resources they might use, events that they respond to etc. The SWS_OS specifies a task state machine that describes the various states of a task and the transitions between these states as shown in Figure 1.

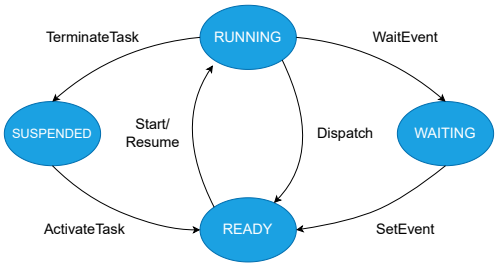

Figure 1: Task State Machine.

The labels of the transitions in Figure 1 are the APIs that are involved in task management. Moreover, the SWS_OS mandates that tasks must have a fixed priority and must execute in a priority order, i.e., among the tasks which are ready to execute (tasks that are in the ready state), the task with the highest priority will execute. This is an important safety property that the implementation must respect.

Initially, for example, at system start-up, all the tasks are in the suspended state. Depending on the configuration, one or more tasks are activated and transit to the ready state. Of these tasks, the task with the highest priority transits to the running state. Furthermore, another task may be activated at a later time that has a priority greater than the currently running task. This new task then causes a preemption where the previous running task transits to the ready state, and the new task transits to the running state. Moreover, the currently running task can wait for events. In this case, the task transits to the waiting state. When the task receives the relevant events, it again transits to the ready state and waits for its chance to transit to the running state. When a task completes its execution, the task terminates and transits to the Suspended state.

The tasks may also need shared resources such as data structures, common variables, communication interfaces, etc., in a mutually exclusive manner. The SWS_OS mandates using the Highest Locker Protocol (HLP) [15] to manage access to shared resources for single-core systems. HLP is a variant of Priority Ceiling Protocol (PCP) [3] where the priority of the task is raised immediately to the ceiling priority of a shared resource upon requesting the shared resource. The task may also acquire the multiple shared resources in a nested manner as shown in Figure 2.

The number of shared resources a task may use must be fixed at design time. The ceiling priority of a shared resource is then equal to the priority of the highest priority task that is configured (or *eligible*) to access the shared resource. The SWS_OS specifies the

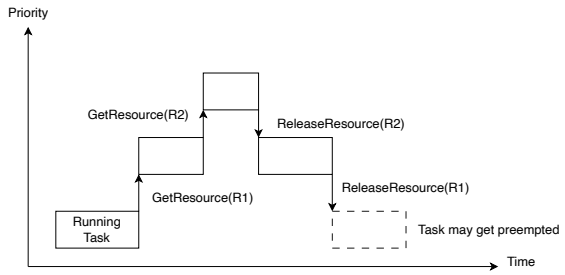

Figure 2: Example nesting of shared resources.

`GetResource` and the `ReleaseResource` APIs for acquiring and releasing a shared resource. It should be noted that nested shared resources must be released in the order opposite to the acquisition order. If the order of release of a nested shared resource is not opposite to acquisition order, it will lead to an invalid system state.

Equipped with the relevant background information presented in this section, we will now describe our approach. The rest of the paper is as follows: In the section 3, we will introduce our approach. In the sections 4 and 5, we will apply our approach to a widely used AUTOSAR-compliant RTOS. We will formalize the requirements of the SWS_OS and then apply the ideas of formal reasoning on the implementation of the relevant APIs in the RTOS. Finally, we discuss some related works in this direction and present some ideas about the future directions of this work in sections 6 and 7, respectively.

3 Our Approach

Our approach consists of 3 steps which are explained below.

1. Formalization of the SWS_OS: In this step, we analyze the SWS_OS to formalize the functional requirements and establish the relevant APIs related to the functional requirements. There are several outcomes of this step. First, a formula in first-order logic is derived which mathematically represents the functional requirement. Second, we pose lemmas which check the consistency of the specification.

The functional requirements are realized by one or more APIs. Moreover, the functional requirements are independent of the implementation of these APIs. This step allows us to formalize the system invariants that a call to an API or a sequence of API calls must satisfy.

To check whether the formalization of a system invariant is correct, we identify characteristic properties of the system invariant. We pose these properties as a lemma for the Dafny theorem prover. Now, the strength of a specification is implied by the number of lemmas the theorem prover can prove automatically. We use this idea to strengthen the formalization of the system invariant in such a way that the theorem prover can automatically prove the required lemmas. However, this might not be the case for all specifications. If the theorem prover is unable to prove the characteristic properties of a specification automatically, we need to prove the lemma using assertion techniques provided by Dafny.

2. Abstraction of the Implementation: In this step, we formalize the source code and derive an abstraction of the relevant APIs that are involved in satisfying the invariants. Identifying the

relevant APIs is relatively easy: the SWS_OS specifies the relevant APIs. However, the challenge is to prove that the abstraction and the source code are equivalent.

We use the calculus of strongest post-conditions (SP) to derive the abstractions (see APPENDIX) [6, 20]. For a given statement α and precondition P, the strongest post-condition, $SP(\alpha)P$, is a proposition that describes the set of states that results from executing α in any pre-state that satisfies P. The calculus provides a set of axioms that allow us to perform symbolic execution. The important property of SP of a statement or a function (which is a sequence of statements) is that the SP is sufficient and a necessary post-condition of a given initial precondition.

We apply the SP calculus to an API to abstractly reason about the effect of each statement in the API without any concern about the implementation details. The calculus gives a proposition that is a conjunction of the post-condition of each statement in the API. It is important to mention here that, as of now, we apply the SP calculus manually to a given API. However, automating this step is possible and we are already working in this direction.

3. Preservation of System Invariants: The analysis in step 1 of our approach derives a proposition that must remain invariant in time. In step 2, we derive an abstraction of the relevant APIs involved in satisfying a functional requirement. In this step, we show that a call to an API or a sequence of API calls preserves the system invariant.

To prove that the system invariants are preserved, we have to show that if an API at the i^{th} time step is invoked in a state where the system invariant holds then at the $(i+1)^{th}$ time step, the call to API or a sequence of API calls lead to a new state where the system invariant is preserved. This is called the invariance lemma.

To automatically prove the invariance lemma, it is necessary to identify the sequence of API calls that establish the invariance. This sequence of calls is the exact behavior of a system that is compliant with the specifications. For instance, as shown in Figure 1, a task which transits to the ready state must be explicitly in the suspended or a waiting state. Moreover, when a task transits to the ready state and enqueued in the ready queue, it may be the case that this task causes a preemption, in which case, the dispatcher must be called. This step allows us to identify such sequences which proves that the APIs satisfy the specifications.

4 Analysis of an AUTOSAR Implementation

We consider the implementation of EB tresos® AutoCoreOS [7], a widely used operating system in the automotive industry, developed by Elektrobit GmbH. We will call it just AutoCoreOS in the remaining paper. AutoCoreOS comes with a configuration and generation tool, EB tresos® Studio, to set up tasks, resources, scheduling tables, etc.

In this paper, we formalize the task state machine and the single-core resource management protocol HLP, as implemented in AutoCoreOS. Due to SWS_OS, it is tractable, albeit not so easy, to identify the required APIs related to task management and the task state machine. However, this step can be automated using approaches such as [21]. AutoCoreOS has many optional debugging and runtime features. These features, such as debug traces, legacy code for compatibility with older applications, etc., are configurable using the Tresos software. We assume that such options are disabled as these do not affect our analysis.

We also assume that the calls to the APIs are *atomic*, i.e., each API either executes to completion or does not execute at all. The APIs are implemented as kernel system calls. To ensure that these system calls are atomic, the usual technique is to disable the interrupts upon entry to the system calls and then re-enable them just before exiting the system call. For simplicity, we assume that the modifications to kernel data structures are atomic.

AutoCoreOS also allows multiple instances (or *activations*) of the same task. This means that a task which is, for instance already in the running state can be activated again. However, the upper bound on the number of instances of a task is configurable at design time and we assume that this upper bound is 1.

With these assumptions in mind, we will now apply our approach to AutoCoreOS.

4.1 The Task Control Block

A task is an executable entity which, together with other tasks of the application, implements the intended logic of the application. The SWS_OS specifies what defines a task. We use this specification to derive a task data type in Dafny as shown below.

The set of tasks in an application is fixed at design time and does not change. We reflect this specification by defining a mathematical set of tasks as a constant. The specification also defines two types of tasks, basic tasks that must eventually terminate and extended tasks that may enter a waiting state and wait for an event.

```
type Context(!new, ==)
datatype TaskType = Basic | Extended
datatype Task = Task(id: nat,
    qPrio: nat,
    events: map<nat, bool>
    taskType: TaskType
    ctx: Context)
const set<Task>
```

Listing 1: Task Datatype.

The *Task* data type in Listing 1 specifies the fields of the task control block (§). The task ID and priority are configured during design time using the Tresos configuration tool. The type of task, extended or basic, is also configured during the design time. In case the type of the task is extended, we can optionally configure events that this task will respond to. The events field in the task control block is a mapping between an event's ID and a boolean value indicating if this particular event is set or not.

An important field of a task structure is the task's context. The context of a task is a set of CPU registers such as the program counter, stack pointer, etc., which holds the current execution environment of the task. Depending on the underlying hardware architecture, managing a task's context may differ greatly. In this work, we do not deal with hardware-level aspects of verification and therefore we abstract the task's context as an uninterpreted data type. However, we specify that this abstract data type can be compared (==) and is not heap based (!*new*).

AutoCoreOS defines two data structures related to the task: a static structure which holds the statically generated configuration

of the task and a dynamic structure which holds the run-time information required for executing the task. For example, the queuing priority of the task is defined statically using Tresos at design time whereas the context of a task is dynamic and changes at run-time.

Moreover, AutoCoreOS has two distinct task types: basic tasks and extended tasks. Basic tasks cannot call the `WaitEvent` system call to enter a waiting state and must call the `TerminateTask` system call, eventually to transit to the suspended state. Extended tasks on the other hand can enter a waiting state and may or may not terminate.

4.2 The Ready Queue

The SWS_OS specifies that the tasks in the ready state must execute in priority order. However, the specification does not describe what algorithm must be used to achieve this. An effective way to address this specification is to implement a priority ordered array or a *sequence* in Dafny. A sequence in Dafny is an immutable data type which represents an ordered list. We specify the ready queue (queue of tasks in the ready state) as a sequence in Dafny. We also specify that this sequence is priority-ordered in Eq. 1.

$$\forall\, i, j :: 0 \leq i < j < |ReadyQueue| \implies Prio(ReadyQueue[i]) \geq Prio(ReadyQueue[j]) \tag{1}$$

An important system invariant for the task state machine is the priority-ordered execution. We can formalize this system invariant using our formalization of the ready queue. In order to specify this system invariant, we observe that at any given time, the ready queue may or may not be empty. If the ready queue is empty, then all the tasks are in the suspended or in the waiting state.

$$ReadyQueue == [] \implies (forallp :: p \in TaskSet \implies State(p) == (Suspended \vee Waiting)) \tag{2}$$

If the ready queue is not empty, then it must be sorted, the tasks in the ready queue are in the ready state, and the task at the head of the queue is the currently running task.

$$ReadyQueue \neq [] \implies \Big(sorted(ReadyQueue) \wedge (\forall p :: p \in TaskSet \wedge State(p) \neq (Suspended \vee Waiting) \wedge p \neq currTask \implies p \in ReadyQueue \wedge State(p) == Ready) \wedge currTask == head(ReadyQueue) \wedge State(currTask) == Running\Big) \tag{3}$$

The system invariant we want to verify is the disjunction ($\vee$) of Eq. (2) and Eq (3). We will refer to this disjunction as `validSM` later in the text.

AutoCoreOS uses a priority-based linked list for its ready queue, ensuring tasks are always sorted by priority, with the highest-priority task at the head and ready to run. When a task is activated, it's placed in the correct position in this list. Our formalization is *functionally* equivalent to AutoCoreOS's approach, as both share the same loop invariants and termination conditions. Since we are not comparing memory efficiency or performance, this functional equivalence is sufficient. However, to reason effectively about linked-list-based algorithms, we need to address the frame problem using the concept of *dynamic frames* in Dafny [10].

The number of tasks is configured and fixed at design time. Therefore, the length of the linked list is fixed. AutoCoreOS implements the ready queue as an array of length equal to the number of tasks. For each task, the "next" pointer in the corresponding task structure points to the next index in the ready queue, where the next higher-priority task is enqueued. The implementation parses the ready queue by following each task node's "next" pointer.

With our formalization of the ready queue as a sequence in Dafny, we ensure that the highest priority task is always at index 0 and the priorities decrease as we go up the sequence. This is represented in Figure 3.

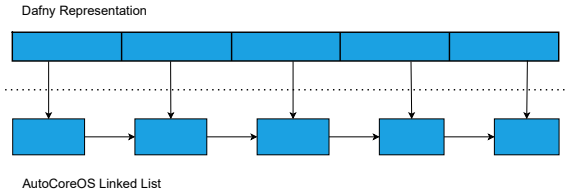

Figure 3: Equivalence between linked list and Dafny sequence.

AutoCoreOS provides an API to enqueue a newly activated task into the ready queue. Using our abstraction, we implement a similar method in Dafny. Dafny methods use *requires* and *ensures* clauses for pre- and post-conditions. Here, the pre-condition ensures the ready queue is sorted before inserting a new task, and the post-condition guarantees it remains sorted after.

The SWS_OS specifies that there can be only one running task and this task is the highest priority task among all the tasks in the ready state. Our specification respects this and we prove that by posing the requirement as a lemma in Dafny (§). This shows that our specification is consistent with the SWS_OS and is indeed a correct interpretation of the specifications.

4.3 The `ActivateTask` System Call

As seen in Figure 1, the `ActivateTask` function transitions a task to the ready state from the suspended state. The `ActivateTask` function also enqueues the task in the ready queue according to its priority. The `ActivateTask` is invoked in a state where `validSM` holds. We first derive an abstraction of `ActivateTask` using the SP calculus. We show a normalized version of the implementation where all optional pieces of code are eliminated (*td* points to the task control block).

```
1  is = OS_IntDisable();
2  ts = td->state;
3  if(ts == OS_TS_SUSPENDED) // Expr_1
4  {
5      td->prio = tp->queuePrio; // Expr_2
6      td->state = OS_TS_NEW; // Expr_3
7      OS_Enqueue(tp); // Expr_4
8  }
```

```
OS_IntRestore(is);
```

Listing 2: Excerpt of ActivateTask Function from AutoCoreOS.

We are focused on the sequence of statements that avoid leading to an error state. Our goal is not to demonstrate that a sequence exists which results in a system call failure and causes the OS to enter an undefined state. Our intention is to show that if a sequence of statements in a system call terminates without an error, then the system call preserves the system invariants. In the case of the ActivateTask system call, the condition guards the criteria that the task must be in the suspended state before it is activated. The "else" part of this condition is a sequence of statements that lead to an error state.

```
P == validSM() //start in a valid state
// init priority of task undefined
Prio(t) == 0
sp(Expr_1)P => State(t) == SUSPENDED // Q1
sp(Expr_2)Q1 => Prio(t) == qPrio // Q2
sp(Expr_3)Q2 => State(t) == READY // Q3
sp(Expr_4)Q3 => ReadyQueue == OS_Enqueue(t)
// Q4
```

Listing 3: Deriving SP for ActivateTask.

As shown in Listing 3, we derive the SP of ActivateTask function using the SP calculus. In Listing 3, State and Prio are abstract functions which represent the state and the priority of the task respectively. These abstract function allow us to reason about the *effect* of the expression without worrying about the implementation details. We obtain the SP of ActivateTask as:

$$\begin{aligned} State(t) == SUSPENDED \ \wedge State(t)' == READY \ \wedge \\ Prio(t) == 0 \ \wedge Prio(t)' == qPrio \ \wedge \\ task \notin ReadyQueue \ \wedge task \in ReadyQueue' \end{aligned} \tag{4}$$

The values of the variables after the assignment are indicated with a "′" symbol. The Eq. (4) is an abstraction of the C code. This proposition clearly shows what is expected after the ActivateTask completes without error. In other words, the proposition represents the behavior of the function.

In a similar manner, we derive abstractions of each system call involved in the task state machine. These system calls are available with further explanations in the artifacts (§).

4.4 The Invariance Lemma

We have our specifications in equations (1) and validSM and an abstraction of our system calls, for instance in (4). We are now ready to derive a sequence of system calls that lead to the invariance of validSM.

We formalize the task state machine based on the ideas in [12]. We define a data type in Dafny that represents the task-state machine as shown below. A complete description of the fields can be found in the artifacts (§).

```
datatype SMState = SMState(
cs: map<Task, TState>,
currTask: Task,
currContext: Context)
```

Listing 4: State Machine Datatype

The SMState data type specifies the task state machine. The data type specifes a mapping between the tasks and their respective states, the currently running task and the context of the currently running task. With this specification of the task state machine, we are able to track the states of each task. We are also able to reason about the state transitions of the currently running task and its context.

The states of the task are encoded as another data type:

```
datatype TState = Ready | Running |
        Suspended | Waiting
```

Listing 5: Task States

Rewriting Eq. (4) in terms of the state machine data type, we formalize the ActivateTask system call as:

$$\begin{aligned} sm.cs[t] == Suspended \ \wedge \\ sm'.cs == sm.cs[t := Ready] \ \wedge \\ sorted(sm.ReadyQueue) \ \wedge \\ sm'.ReadyQueue == \\ enqueueTask(t, sm.ReadyQueue) \ \wedge \\ sorted(sm'.ReadyQueue) \ \wedge \\ sm'.currTask == getHead(sm'.ReadyQueue) \end{aligned} \tag{5}$$

Here, the new state of the state machine is represented as sm'. The function enqueueTask is the function that uses our formalization of the ready queue to insert a task in an appropriate position in the ready queue. It is clear that Eq. (4) is semantically equivalent to Eq. (5). Similar representations are implemented for other system calls involved in the task state machine (§).

Now, we are in a position to prove that the system invariant is preserved as a task transits through the various states of the state machine. In this regard, we first pose the invariance lemma (§).

Lemma 4.1. *Whenever the state machine is invoked in a state where the system invariant is valid and a task transits to a new state, the system invariant is preserved in the new state.*

Clearly, only Eq. (5) by itself cannot satisfy this lemma. This is so because the task which is enqueued may have a priority higher than the task which is currently at the head of the queue and hence running. In this case, a preemption occurs before the control is handed back to the application. Similar situation arises when a task terminates and is removed from the queue or when a task is activated because it received the event it was waiting for. We show here the sequence of steps that lead to invariance for basic tasks.

We show an abstraction of the task dispatcher as implemented in AutoCoreOS. The task dispatcher is invoked from the hardware-specific context management part of AutoCoreOS. Since we have abstracted the task's context, we do not consider context management here. However, we specify that the context manager invokes the dispatcher if, after the activate system call completes, the head of the updated ready queue and the currently running task are not

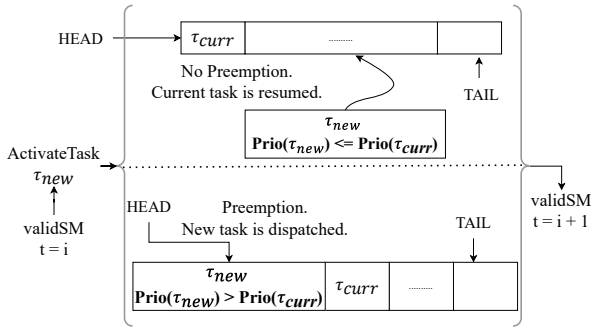

Figure 4: Invariance of `ActivateTask`.

the same. Moreover, the context management in AutoCoreOS ensures that if the head of the updated ready queue and the currently running task are the same, the task is resumed (Figure 4).

In our abstraction, we have specified this behavior in two distinct predicates (§). The task t is the new task that was activated. We show here the abstraction of the dispatch function as implemented in AutoCoreOS in the `OS_Dispatch` system call.

$$\begin{aligned} sm'.currTask.context == sm.currContext \wedge \\ sm'.cs == sm.cs[sm.currTask := Ready] \wedge \\ sm'.cs == sm.cs[t := Running] \wedge \\ sm'.currTask == t \wedge \\ sm'.currContext == t.context \end{aligned} \tag{6}$$

The task dispatcher is invoked in the state obtained after the `ActivateTask` system call completes. The final state of the state machine is then a **conjunction** of Eq. (5) and Eq. (6). **This behavior is a consequence of the specification of the `ActivateTask` and the `OS_Dispatch` system calls**. Moreover, the same behavior is also observed in AutoCoreOS. This means thatAutoCoreOS preserves the `validSM` system invariant in the case where the activated task causes a preemption.

Encoding such sequences of API calls in Dafny is not trivial. In Listing 4, the `cs` field in the state machine data type maps a task to it's state. This can be viewed as each task having its own instance of the state machine from Figure 1. We therefore introduce two type functions: $Trace : \mathbb{N}^+ \rightarrow SMState$ and $Schedule : \mathbb{N}^+ \rightarrow Task$ to map a time step to the state of the task state machine and a task [12] (§). Since, each task has its own instance of the state machine, the two type functions allow us to model the transitions of a random task from the task set between the states.

```
lemma Invariance(trace: Trace,
    schedule: Schedule)
    requires isSchedule(schedule)
    ensures Init(trace(0)) ==>
        validSM(trace(0))
    ensures forall i: nat ::
        validSM(trace(i)) &&
  Transitions(trace(i), schedule(i),
    trace(i + 1)) ==>
      validSM(trace(i + 1))
```

Listing 6: Invariance Lemma

In the lemma, `Transitions` is a predicate that returns true if any one of the transitions holds true (§). All the possible transitions are encoded in this predicate.

As we have seen, we have used our approach to formally specify the task state machine and prove that the abstraction of the implementation always respects the specifications. The abstraction of the implementation is equivalent to the actual implementation, we conclude that the actual implementation also respects the specifications.

We now move on to the specifications of HLP and then abstract the implementation in AutoCoreOS.

5 The Highest Locker Protocol

We described HLP in section 2 and now apply our approach to specify and verify the implementation in AutoCoreOS. Based on the specifications of a shared resource in the SWS_OS, we create a resource data type in Dafny. The SWS_OS also specifies the state of a resource with respect to the set of tasks eligible to access the resource. A resource can be in two distinct states: acquired or released. Moreover, the set of shared resource is statically defined at design time and is constant throughout the life time of the OS. These requirements are formalized in Dafny as shown:

```
datatype ResState = Acquired | Released
datatype Resource = Resource(
    eligibleTasks: set<Task>,
    ceilingPrio: nat,
    prevPrio: nat,
    resId: nat)
const Resources: set<Resource>
```

Listing 7: Resource Datatype

The ceiling priority of a shared resource is equal to the highest priority out of all the tasks that are eligible to access the shared resource. Mutual exclusion is guaranteed by HLP due to the ceiling priority of the resource. However, the actual functional requirement of a shared resource management protocol is a independent of the implementation. We specify the property of mutual exclusion next.

5.1 Mutual Exclusion

The property of the mutual exclusion states that if an eligible task has acquired a shared resource, then no other eligible task is allowed to acquire the same shared resource. For a given resource (let's say *res* of type `Resource`) this property can be formally specified as follows (§).

$$\begin{aligned} validSM() \ \wedge \forall p :: p \in res.eligibleTasks \wedge \\ ResourceState(res, p) \neq Released \implies \\ \nexists q :: q \in res.eligibleTasks \ \wedge \ p \neq q \wedge \\ ResourceState(res, q) \neq Released \end{aligned} \tag{7}$$

Since we treat the shared resource management protocol as a state machine, the Eq. (7) represents the system invariant for the protocol. To verify that our specification for mutual exclusion is

correct, we pose a lemma in Dafny (§). The lemma assumes that there are two tasks both eligible to access a given shared resource and have acquired the shared resource. If the specification of mutual exclusion property is correct, then it means both the tasks in our assumption are actually the same task. Given the strength of our specification, the theorem prover is able to prove automatically that this indeed is the case.

5.2 Release Order

A task may have acquired several shared resources in a nested manner. The specifications of HLP mandate that the order in which the shared resources are released is opposite to the order in which they were acquired, i.e., the last nested shared resource acquired is the first to be released. This specification motivates the use of a stack-like mechanism to track the nesting of shared resources.

We formalize a stack in Dafny using a sequence. We implement the push and pop operations which are characteristic operations of a stack. Moreover, this specification calls for an updated task structure that we already formalized in section 4. We add a sequence to the task structure where each element of the sequence is a shared resource. Whenever a task acquires a shared resource, the shared resource is pushed onto this sequence and whenever the task releases the shared resource, the shared resource is popped from this sequence. The updated task structure is shown below.

```
datatype Task = Task(id: nat,
    qPrio: nat,
    events: map<nat, bool>
    taskType: TaskType
    lockSeq: seq<Resource>
    ctx: Context)
```

Listing 8: Updated Task Structure

```
function Push(lockSeq: seq<Resource>,
    res: Resource): seq<Resource>
    requires res in Resources
    ensures Push(lockSeq, res) != []
{[res] + lockSeq}

function Pop(lockSeq: seq<Resource>):
    (Resource, seq<Resource>)
    requires lockSeq != []
{(lockSeq[0], lockSeq[1..])}
```

Listing 9: Resource Stack Management Functions

5.3 HLP APIs

We specify the HLP as a state machine. We specify another data type similar to `SMState` to track the state transitions of a task which acquires a shared resource and eventually releases it (§).

```
datatype RMState = RMState(
    cs: map<Task, ResState>,
    sm: SMState,
    occupiedBy: Task)
```

Listing 10: Resource State Manager

As we have seen in Figure 2, there are 2 APIs related to the protocol: `GetResource` and `ReleaseResource`. We now analyze these APIs using the step 2 of our approach to derive an abstraction from the implementation in the AutoCoreOS.

It is important to note that the APIs related to the shared resource management are C functions. This implies that for a task to acquire or release a shared resource, the task must be in the running state. Since there can be only one running task in the system as per the specifications of the task state machine and moreover, we have also assumed that the interrupts are disabled upon entering a system call, it implies that only the currently running task can acquire or release a shared resource. For HLP, it is therefore required that the task state machine is in a valid state (`validSM`).

We show here the abstractions of the acquisition (Eq. (8)), and releasing (Eq. (9)) a shared resource.

$$\begin{aligned} \forall\, p \in res.eligibleTasks \implies \\ ResourceState(res, t) == Released \wedge \\ t == currTask \ \wedge\ t.rPrio == res.prioCeiling \wedge \\ rm'.occupiedBy == t \wedge \\ rm'.occupiedBy.lockSeq == Push(res) \wedge \\ ResourceState(res, t)' == Acquired \end{aligned} \tag{8}$$

$$\begin{aligned} t == currTask \ \wedge rm.occupiedBy == t \wedge \\ rm.occupiedBy.lockSeq \neq [] \wedge \\ ResourceState(res, t) == Acquired \wedge \\ t.rPrio == res.prevPrio \wedge \\ ResourceState(res, t)' == Released \wedge \\ rm'.occupiedBy.lockSeq == Pop() \end{aligned} \tag{9}$$

AutoCoreOS uses a stack based on a linked list to track nested calls to `GetResource`. We have already shown an equivalence between a linked-list and a Dafny sequence in Figure 3. Our abstraction of a stack and the implementation in AutoCoreOS is equivalent in the sense that the pre- and post-conditions are the same. Moreover, if a task calls `ReleaseResource` without calling `GetResource` first, it will lead to an error. The AutoCoreOS checks if the last lock acquired is the lock that is being released. Our specification of the locking sequence as a stack ensures this behavior is reflected in our abstraction.

5.4 Invariance Lemma for HLP

Similar to the invariance lemma of the task state machine, we must show that as a task transiting through the various states of the shared resource state machine must preserve the invariance of Eq. (7). In this case, we track the states of the resource state machine using another typed function which returns the state of the shared resource state machine at each time step (§).

The invariance lemma for HLP shows that if an eligible task acquires or releases a shared resource, Eq. (7) is preserved. Moreover, it also states that the task that invoked the `GetResource` or `ReleaseResource` system calls is the currently executing task. Given the strength of our specifications and the abstractions of

the relevant system calls, the Dafny theorem prover automatically proves this lemma.

An important property of HLP is that whenever a task acquires a shared resource, the priority of the task is raised to the ceiling priority of the shared resource. This provides mutual exclusion between tasks that require the same shared resource. However, this does not affect the priority-ordered execution of the tasks. Any task that has a priority greater than the ceiling priority of the currently acquired shared resource can cause a preemption. The invariance lemma also ensures that when a task invokes the shared resource state machine in a state where `validSM` and Eq. (7) holds, then after the transition, `validSM` and Eq. (7) still holds.

With our approach we have (1) formally specified the HLP shared resource management protocol, (2) Derived the abstractions of the APIs implemented in AutoCoreOS, and (3) proved that the implementation respects the specifications through time.

6 Related Work

Fengwei Xu et al. [19] propose a practical verification framework for preemptive OS kernels that models the correctness of API implementations as contextual refinement of their abstract specifications. They have applied their approach to the μC/OS-II to verify important properties such as freedom from priority inversion. The authors model kernel correctness as contextual refinement between concrete implementation and abstract specifications. The authors provide a specification language for defining high-level abstract models of OS kernels and develop a program logic for refinement verification of concurrent kernel code with multilevel interrupts. Verification is automated using automated tactics with the Coq theorem prover.

Our work differs from [19] in the way in which the abstraction of a concrete implementation is derived. The authors have developed a specification language for abstracting the OS kernels whereas we use the SP calculus to derive the abstraction. Using a contextual refinement approach, the authors prove the equivalence between the concrete implementation and the abstraction. In contrast, we use a more direct approach and reason about the *effect* of each expression in a given system call to derive a formula in first-order logic using SP calculus.

In [14], Batista Ribeiro et al. use the formal modeling tool UPPAAL to formally verify OS models that can be used as a foundation for verifying compositional applications. The authors divide the UPPAAL models in three layers: the application, the kernel interface, and the OS model. The authors also introduce the concept of abstract tasks to cover all possible task behaviors and ensure the OS model is verified against all possible scenarios.

A limitation of [14] lies in the fact that the OS models do not guarantee implementation correctness, and the full path from OS models to C code must be verified. We overcome this limitation by using SP calculus to prove an equivalence between a concrete implementation and the abstraction we used in our mathematical model.

Formal reasoning techniques such as predicate transformers (SP calculus, for instance) were also used in [17]. The paper proposes a novel technique called Strongarm for inferring concise post-conditions for API methods using symbolic execution and predicate transformers. The technique was applied to popular Java libraries to infer concise specifications smaller than those derived using conventional SP calculus.

Our approach uses conventional SP calculus, but does not reduce the derived specifications to a more concise form. This has motivated us towards similar techniques that could be useful in obtaining more concise abstractions of the implementation without losing the semantics of the implementation.

The IronFleet Framework [9] uses Dafny to demonstrate the feasibility of mechanically verifying practical distributed system pimplementations that match simple, centralized specifications. IronFleet is the first system to mechanically verify the liveness properties of a practical distributed protocol and implementation. It includes three main components: Dafny, a verification-aware programming language; IronRSL, a high-level specification language for distributed systems; and IronFleet Tools, a suite of utilities for managing and verifying system specifications. This comprehensive framework enables rigorous correctness guarantees, improving the reliability of distributed systems.

Although the IronFleet framework provides rigorous correctness guarantees, the correctness is not absolute and relies on several assumptions. Moreover, the verification requires significant development efforts even using automated tools. Furthermore, the framework focuses on verifying newly written code in a verification-friendly language rather than verifying existing code.

We demonstrate using our approach on existing code. Moreover, the development effort is minimal as the SP of a given system call can be formulated quite fast and efficiently. Although we have yet to implement an automatic SP generator, we are progressing strongly in this direction. Furthermore, the assumptions we have made are not stringent and reflect a real system. For instance, our assumption that the system calls are atomic is also reflected in the AutoCoreOS which disables the interrupts upon entry to a system call and enables them again upon exit.

We have borrowed several ideas from [12]. Leino specifies a ticketing system and verifies the liveness and mutual exclusion using Dafny. However, the specification is not a representation of a concrete implementation. The ticketing system is specified as a state machine. Leino uses an abstract process and also specifies the various "APIs" that *could* represent a real implementation. In reality, a simple ticketing system may not be sufficient in a complex OS such as EB tresos AutoCore OS.

We use the approach to specify the task scheduling and the shared resource management in AUTOSAR as a state machine. However, we refer to a concrete implementation and verify that the implementation respects the specifications. The idea of using the invariance lemma was the main motivation of our approach.

In [13] Matias et al. use Dafny to specify and verify the implementation of FreeRTOS. The specification was created based on the FreeRTOS API documentation and an existing Z model, and the implementation was a mix of code from the existing FreeRTOS codebase and code constructed solely from the Dafny specification. The Dafny code captures the data structures, algorithms, and scheduler operations and documents the required behavior and constraints. However, there is no discussion about system invariants and proof that the implemented APIs respect the system invariants.

7 Conclusion and Future Work

In this paper, we present an approach to analyze AUTOSAR specifications and show that a given implementation of an OS respects the specifications. Our analysis of the SWS_OS yielded specifications in the form of a formula in first-order logic. These formulas represent the system invariants that are independent of the implementation. Although we derived these formulas manually based on our understanding of the SWS_OS, the characteristic lemmas are proven automatically by the Dafny theorem prover showing that our interpretation of the specifications is correct.

We used SP calculus to derive an abstraction of the implementation. This abstraction is another formula in first-order logic where each conjunct represents the strongest post-condition of each statement of the API under consideration. Hence, the abstraction is equivalent to the implementation.

We do not consider parts of the implementation that lead to an error state, as reaching such a state implies that system invariants are already violated. Our focus is on control flows that avoid error states. However, avoiding an error state alone does not guarantee that the system invariants are satisfied. Instead, for control flows that do not reach an error state, we demonstrate that the implementation satisfies the system invariants using the invariance lemma.

In the artifacts provided, we have also added a "wrong" lemma for every lemma. The wrong lemma tries to prove a wrong conclusion of our formalization of the specification. For instance, in the invariance lemma in Listing 6, the formalization would be wrong if both `validSM` and its negation holds. If such a case arises, then either our formalization of the specification and/or the abstraction of the implementation is incorrect. This lets us confidently conclude that our abstraction and formalization approach is correct.

Our proposed methodology focuses on AUTOSAR-compliant RTOS. However, the formal reasoning approach can be extended to other standards, such as POSIX. SP calculus is independent of the underlying RTOS and we can reason about the *effects* of the statements of a given API from a POSIX-compliant RTOS in the same way we have presented in this paper. The differences may arise due to different functional (or non-functional) requirements of other standards.

An important missing link in our approach is automatically generating the strongest post-conditions. Since the code base we had to analyze was relatively small, it was not a problem. However,AutoCoreOS has many APIs that are directly or indirectly involved in many functional requirements of the SWS_OS. It will be intractable to analyze and derive the abstractions of each API manually. It is desirable, therefore, to have a way to automatically generate the strongest post-conditions. In this direction, we are working on a strongest post-condition generator written using Z3 and Python [5]. The generator will be able to generate the strongest post-conditions using the SP calculus for complex code constructs such as loops, conditions, function calls, etc.

We have also not considered the hardware-level aspects of the OS. For instance, AutoCoreOS has a sophisticated context-switching mechanism where the context registers of the outgoing task are saved and that of the incoming task is loaded. Based on the hardware platform, the context-switching mechanism may change. We therefore abstract the context of a task as an abstract data type in Dafny. This allows us to use simple comparison and update of a task's context without the need of hardware-level details. However, hardware-level formal specification and verification is beyond the scope of this paper. A survey on assertion-based hardware verification can be found in [18].

We also aim to identify and formalize more functional requirements related to safety and consistency of the AutoCoreOS for example, in a multi-core setup. Ongoing work aims to extend this paper with important functional requirements in a multi-core setup. This will require formalizing important aspects of multi-core systems such as inter-core shared resource management and showing that the relevant APIs satisfy the requirements.

APPENDIX

Dijkstra introduced predicate transformers [6], focusing on weakest preconditions, which ensure a post-condition holds. In contrast, strongest post-conditions describe what can be inferred after a program runs. Expressed in first-order logic, strongest post-conditions allow formal reasoning about code. We use this approach to derive an abstraction of C code, transforming it into an equivalent first-order logic formula. The calculus is provided assuming a given precondition P.

(1) Sequence axiom

$$sp(\alpha;\beta)P := sp(\beta)\,(sp(\alpha)P)$$

(2) Assignment axiom

$$sp(x := e)P := \exists x'.x = e(x') \ \wedge \ P(x')$$

(3) Nondeterministic choice

$$sp(\alpha \cup \beta) := sp(\alpha)P \ \vee \ sp(\beta)P$$

(4) If-then-else constructs

$$sp(if\ Q\ \alpha\ \beta)P := sp(\alpha)(Q \ \wedge \ P) \vee sp(\beta)(Q \ \wedge \ \neg P)$$

(5) Loops

$$sp(\alpha^*)P = p \ \vee \ sp(\alpha^*)\,(sp(\alpha)P)$$

For programs with loops, termination must be proven explicitly.

Acknowledgments

This work was supported by (1) the FFG under contract No.881844: "Pro²Future" (Products and Production Systems of the Future), Graz University of Technology (TU Graz), and Elektrobit Automotive GmbH in the joint research project CompEAS and (2) TU Graz Open Access Publishing Fund.

References

[1] AUTOSAR. 2022. Software Specification od Operating System. (2022). https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_OS.pdf
[2] S. Beringer and H. Wehrheim. 2016. Verification of AUTOSAR Software Architectures with Timed Automata. In *Critical Systems: Formal Methods and Automated Verification.*
[3] A. Burns. 2013. The Application of the Original Priority Ceiling Protocol to Mixed Criticality Systems.
[4] J.-L. Béchennec, O. H. Roux, and T. Tigori. 2018. Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In *CoDIT'18.*
[5] L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems.*

[6] Edsger Wybe Dijkstra. 1997. *A Discipline of Programming* (1st ed.). Prentice Hall PTR, USA.
[7] Elektrobit. [n. d.]. Elektrobit tresos® AutoCoreOS. https://www.elektrobit.com/products/ecu/eb-tresos/bsw/
[8] L. Fang, T. Kitamura, T. B. Ngoc Do, and H. Ohsaki. 2012. Formal Model-Based Test for AUTOSAR Multicore RTOS. In *Intl. Conf. Software Testing, Verification and Validation*.
[9] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. 2015. IronFleet: proving practical distributed systems correct. *Proc. SOSP* (2015).
[10] I. Kassios. 2011. The dynamic frames theory. *Formal Aspects of Computing* (2011).
[11] K. R. M. Leino. 2010. Dafny: an automatic program verifier for functional correctness *(LPAR)*.
[12] K. Rustan M. Leino. 2017. Modeling Concurrency in Dafny. In *International School on Engineering Trustworthy Software Systems*.
[13] Matthew John Matias. 2014. Program Verification of FreeRTOS using Microsoft Dafny.
[14] L. Batista Ribeiro, F. Lorber, U. Nyman, K. G. Larsen, and M. Baunach. 2023. A Modeling Concept for Formal Verification of OS-Based Compositional Software. In *FASE'23*.
[15] L. R. Sha, R. R. Rajkumar, and J. P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers* (1990).
[16] Sebastian Siegl, Kai-Steffen Hielscher, Reinhard German, and Christian Berger. 2011. Formal specification and systematic model-driven testing of embedded automotive systems. In *DATE'11*.
[17] J. L. Singleton, G. T. Leavens, H. Rajan, and D. R. Cok. 2019. Inferring Concise Specifications of APIs. *ArXiv* (2019).
[18] H. Witharana, Y. Lyu, S. Charles, and P. Mishra. 2022. A survey on assertion-based hardware verification. *ACM Computing Surveys (CSUR)* (2022).
[19] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *CAV'16*.
[20] L. Zhang and B. L. Kaminski. 2022. Quantitative strongest post: a calculus for reasoning about the flow of quantitative information. *Proc. ACM Program. Lang.* OOPSLA1 (2022).
[21] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *Intl. Conf. Automated Software Engineering*.

Received 14 August 2024; accepted 07 October 2024; revised 28 October 2024