Formal Verification of Resource Synchronization Protocol Implementations: A Case Study in RTEMS

Junjie Shi, *Student Member, IEEE*, Christoph-Cordt von Egidy, Kuan-Hsun Chen, *Member, IEEE*, and Jian-Jia Chen, *Senior Member, IEEE*

***Abstract*—To avoid race conditions and ensure data integrity, resource synchronization protocols have been widely studied in real-time systems for decades, providing systematical policies to guarantee a bound on priority inversion-induced blocking time and the avoidance of deadlocks. However, the corresponding realization is often based on assumed abstractions and necessary adaptions in a real-time operating system, by which the theoretically proven properties of such a protocol may not be delivered, leading to potential mismatches. To prevent such mismatches, in this work, we propose to contract the obligations of involved primitives and operations, and apply the deductive verification on a corresponding implementation. To this end, we present a modularized verification framework and demonstrate its applicability by verifying the official implementation of the immediate ceiling priority protocol (ICPP) and the multiprocessor resource sharing protocol (MrsP) in RTEMS, resulting in the discovery of long-stayed mismatches for both synchronization protocols. To resolve them, we provide a possible remedy for the ICPP and an additional precondition regarding nested locking for the MrsP.**

***Index Terms*—Formal verification, real-time operating system (RTOSe), resource synchronization protocol.**

I. Introduction

In real-time systems, concurrent tasks may access shared resources, such as shared data, files, and memory. The code segment that a task executes while accessing the shared resource(s) is called *critical section*, which is protected by binary semaphores or mutex locks provided by the operating system (OS). To prevent race conditions, accesses to shared resources are mutually exclusive. In the presence of critical sections, some problems might break the timeliness, leading to unbounded worst-case response time (WCRT): 1) priority inversion happens when a job with higher priority is blocked (indirectly) by one or multiple other jobs with lower priority, among which the job with the lowest priority has obtained the same requested shared resource earlier and 2) deadlock can happen when two tasks request the same (at least) two shared resources in a different order simultaneously, and are therefore waiting for each other circularly.

To prevent the above problems, resource synchronization protocols have been widely studied since the 1990 s. In uniprocessor real-time systems, the priority inheritance protocol (PIP) and the priority ceiling protocol (PCP) were proposed by Sha *et al.* [37] and the stack resource policy (SRP) was proposed by Baker [8]. The immediate ceiling priority protocol (ICPP), has been widely applied in real-time systems, e.g., in Ada as ceiling locking and POSIX as priority protect protocol. Along with the demand of computational power, several multiprocessor resource synchronization protocols also have been proposed, such as the distributed PCP (DPCP) [36], the multiprocessor PCP (MPCP) [35], the multiprocessor resource sharing protocol (MrsP) [13], and the dependency graph approach (DGA) [18], [38].

Such a protocol can be formally described as a set of rules that are composed of abstracted system models, under certain assumptions at the OS level. However, these abstractions and assumptions may not always hold while realizing a protocol. For example, some OSes may not allow multiple tasks with the same priority, where the inherited ceiling priority, e.g., in ICPP, has to be compensated by setting ceiling priorities differently and excluding these priorities from regular priorities that have been assigned to other tasks [31]. Some additional components, i.e., the helper mechanism in MrsP and the busy waiting inside a FIFO-queue, require appropriate data structures and operations in the target OS [16]. Any necessary adaption due to the constraint imposed by the OS in practice may lead to a mismatch to the original specifications, resulting in unexpected consequences.

To this end, several prominent works have been proposed to formally verify the real-time OSs (RTOSes). The *seL4* is proposed as a high-assurance and high-performance microkernel [30], which has been entirely formally verified against its abstracted specifications. Gu *et al.* [28] proposed CertiKOS, where an architecture for concurrent OS kernel is verified

Manuscript received 27 July 2022; accepted 28 July 2022. Date of publication 9 August 2022; date of current version 24 October 2022. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG), as part of the Collaborative Research Center under Grant SFB876, subproject A1 and A3 (http://sfb876.tu-dortmund.de/); and in part by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme through the Project PropRT under Agreement 865170. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding author: Junjie Shi.)*

Junjie Shi, Christoph-Cordt von Egidy, and Jian-Jia Chen are with the Design Automation for Embedded Systems Group, TU Dortmund University, 44227 Dortmund, Germany (e-mail: junjie.shi@tu-dortmund.de; cordt.von-egidy@tu-dortmund.de; jian-jia.chen@tu-dortmund.de).

Kuan-Hsun Chen is with the Chair of Computer Architecture and Embedded Systems, University of Twente, 7500 AE Enschede, The Netherlands (e-mail: k.h.chen@utwente.nl).

Digital Object Identifier 10.1109/TCAD.2022.3197501

layer-wisely from the bottom up. An automatic verification approach is also proposed, by checking all possibly reachable states of registers and memory automatically [34]. However, most approaches only study the verification of RTOSes, whilst how to verify the implementations of resource synchronization protocols still lacks research.

Our Contribution: In this work, we focus on the formal verification of synchronization protocols implemented in an RTOS by assuming the underlying functionalities are correct. The contributions in a nutshell are as follows.

1) We propose to contract the obligations of involved primitives and operations, and apply the deductive verification on the corresponding implementation of a protocol in a targeted OS (see Section III).
2) We present a framework for formally verifying the properties of the synchronization protocol implementations in a given OS (see Section IV).
3) We provide two case studies to verify ICPP (Section V) and MrsP (Section VI), implemented in the official RTEMS. In consequence, we discover long-stayed mismatches and provide possible remedies for both protocols. The corresponding source code can be reviewed in [4].

II. Basic Rules of Synchronization Protocols

Resource synchronization protocols are defined as a set of rules that each task has to follow during resource accesses. In the following, we show their basic rules.

Scheduling Algorithms: A synchronization protocol has to define the supported scheduling algorithms, either earliest deadline first (EDF) or fixed priority (FP). When EDF is applied, the job with the earliest absolute deadline has the highest priority, whereas the priorities for all tasks are predefined when FP is applied. For multiprocessor systems, under global schedule, all tasks are dispatched dynamically over different processors. In contrast, each task is assigned statically on a certain processor in advance under partitioned schedule, no migration is allowed.

Request Ordering: The order of concurrent requests for the same shared resource also has to be determined. When two or more tasks are blocked by the same shared resource, the waiting queue of these tasks has to be sequenced by a certain policy. Two common policies are FIFO queue and priority-based queue. In a FIFO queue, tasks are ordered by the requesting time, the task with earlier requesting time can acquire the corresponding shared resource earlier, which bounds the maximum waiting time for each task. In a priority-based queue, tasks are ordered by their current priorities, which are normally the tasks' scheduling priorities. Additionally, there is a third policy which is introduced by Shi *et al.* [38], where the access order for each shared resource is predefined in a job level for all tasks within one hyper period. Therefore, the wait queue is sequenced by the predefined access order.

Waiting Mechanism: The semantic, i.e., how a task is waiting for an occupied resource, has to be identified. Under a suspension-based synchronization protocol, a task that is waiting for accessing to a currently unavailable shared resource is suspended by adding itself into a wait queue. Under a spin-based protocol, the task does not give up its privilege on the current processor. It executes a spinning loop and continuously checks if the requested resource is available until it can access to the requested resource and start its critical section.

Bound Measure: The measure to bound the maximum blocking time and prevent unbounded priority inversions has to be set up. Under nonpreemptive execution, once a task starts its critical section, it cannot be preempted by any of other tasks in this processor regardless of their priorities and deadlines. Similarly, priority boosting allocates a boosted priority to each critical section, which is higher than the highest regular priority for scheduling of all tasks. However, under priority boosting, a critical section can still be potentially preempted by another critical section with higher boosted priority. The other promising approach is priority ceiling: When a task executes its critical section, the priority can be prompted to the corresponding resource's ceiling priority, where the ceiling priority can be determined either statically or dynamically. When the ceiling priority of a shared resource is defined statically, it simply equals to the highest priority of any task that may request the resource. If dynamic ceiling priority is applied, the ceiling priority is defined as the highest priority of all tasks that are currently locking or will lock the shared resource, i.e., tasks in the corresponding waiting queue.

Execution Place: Different to the protocols for uni-processor systems, the places where to execute the critical sections of a task have to be determined for a multiprocessor resource synchronization protocol, either locally or remotely. For the former, the critical sections of a task can be executed along with its noncritical sections on the processor where the task is currently assigned. For the latter, critical sections are executed on specified processor(s) where the corresponding resources are assigned on. In some protocols, the local executed critical sections can also be executed remotely. For example, the *help mechanism* in MrsP allows the current resource owner to execute its preempted critical section on a remote processor, where a task is spin waiting for the same resource.

III. Deductive Verification via Contracts

Although each protocol contains several rules that can be implemented, the combination of these rules is complicated when all the details have to be considered, e.g., the order of priority modifications, the queue-based operations, and illegal inputs checking. One approach is to *test* sufficient inputs and validates the derived outputs. However, a sufficient test set that covers all possible situations is difficult to be derived, especially for multiprocessor systems. The execution of a test case can only *validate* the behavior of the whole systems, including primitives from the OS's kernel, hardware-specific code, and the employed hardware or simulation platform. Any observed error can also be caused by the interplay of these low leveled components. Hence, it is difficult to pinpoint the real issues in the implementation of the protocol itself.

In fact, the formal descriptions of a protocol are based on abstracting from OS- or hardware-specific details. Instead of validation, all the properties that are desired to be achieved by

```
int absDiv(int x, int y){
  int d1, d2, res;
  if (x >= 0) {d1 = x;}
  else {d1 = x * −1;}
  if (y > 0) {d2 = y;}
  else if (y < 0) {d2 = y * −1;}
  else {return −1;}
  res = d1 / d2;
  return res;
}
```

Listing 1. One implementation of the function absDiv.

a resource synchronization protocol can be formally proven (so-called *verification*) based on its formal descriptions. One approach is based on *model checking*. The considered system is first modeled in a formal language where all the required properties are specified in logic formulas, by which a model checker can be applied to automatically check the property specifications. However, the system model is difficult to be specified and there is no guarantee of correctness. To overcome the drawbacks of above approaches, *Deductive Verification* [24], [29] is applied in this work to verify the properties of a given implementation for a resource synchronization protocol directly.

A. Hoare Triple and Weakest Precondition

To specify a certain property of a program, a *Hoare Triple* of the form {*Pre*}*program*{*Post*} is formulated, where the postcondition *Post* holds if the *program* is executed with fulfilled precondition *Pre*. If a postcondition is supposed to hold in any possible case, the precondition is just *true*. To check such a property against a program, its weakest precondition (wp) that is required to satisfy the postcondition is derived. If the defined precondition *Pre* implies the derived wp, the property is proven to hold for the analyzed program. The development of the wp is often performed backward through the code by iteratively transforming the postcondition based on the code statements using the rules defined by Garoche [26].

We provide an example, considering a function `absDiv(x,y)` that divides $|x|$ by $|y|$ in Listing 1. Since a division by zero is illegal (function returns -1), the desired behavior (*Property*) can be formulated as follows, where \res is the value returned by `absDiv`:

$$\{y \neq 0\}\text{absDiv}\{\backslash\text{res} = |x|/|y|\}. \tag{1}$$

Furthermore, the result is expected to be non-negative since both divisor and dividend are with non-negative values. Therefore, a new *Property* can be formulated

$$\{y \neq 0\}\text{absDiv}\{\backslash\text{res} \geq 0\}. \tag{2}$$

Statement 3 based on a implementation in Listing 1 concludes the derivation of the wp for Property 1

$$\begin{aligned} x \geq 0 \Rightarrow & \left[\left(y > 0 \Rightarrow \frac{x}{y} = \frac{|x|}{|y|}\right) \wedge \left(y < 0 \Rightarrow \frac{x}{-y} = \frac{|x|}{|y|}\right) \wedge \right.\\ & \left.\left(y = 0 \Rightarrow -1 = \frac{|x|}{|y|}\right)\right] \wedge \\ x < 0 \Rightarrow & \left[\left(y > 0 \Rightarrow \frac{-x}{y} = \frac{|x|}{|y|}\right) \wedge \left(y < 0 \Rightarrow \frac{-x}{-y} = \frac{|x|}{|y|}\right) \wedge \right.\\ & \left.\left(y = 0 \Rightarrow -1 = \frac{|x|}{|y|}\right)\right]. \end{aligned} \tag{3}$$

```
/*@ // auxiliary predicates and logic_functions
  predicate IsZero(int x) = x == 0;
  predicate NonZero(int x) = ! IsZero(x);
  logic int abs_div(int x, int y) = \abs(x) / \abs(y);
*/
/*@ // function contract
  behavior err:
    requires IsZero(y);
    ensures \result == −1;
  behavior div:
    requires NonZero(y);
    ensures \result >= 0;
    ensures \result == abs_div(x,y);
  complete behaviors;
  disjoint behaviors;
*/
int absDiv(int x, int y) {...}
```

Listing 2. Example of ACSL function contract using predicates, logic functions, and builtin functions (\abs).

By replacing the $= (|x|/|y|)$ with ≥ 0, the wp for Property 2 can be obtained as well.

B. ACSL Function Contracts and Frama-C

To verify a function's specification consisting of Hoare-triples, also called *function contract*, against the source code of the implementation of a targeted protocol, Frama-C can be applied [2]. The plugin wp of Frama-C provides the capabilities for static analysis and deductive verification of source code. The contracts are formulated by using the ANSI/ISO C specification language (ACSL) [10]. It allows to formally specify the behavior(s) of a function as a function contract in the form of annotations to its source code enclosed in special comments, i.e., `//@` or `/*@··· */`.

Contracts can consist of different *behaviors*, each of which ensures a set of postconditions depending on different preconditions or assumptions and may be declared to be *complete* or *disjoint*. To ease the formulation of a specification, constructs like *predicates*, *logic functions*, and *assertions* are provided. A predicate evaluates received parameters to either `true` or `false`. Predicates can be used within assertions, function contracts, or other predicates. Logic functions can have any return type and perform assignments or computations. An ACSL function contract for the previous example is given in Listing 2. It describes the Properties 1 and 2. The behaviors are declared to be disjoint, i.e., no two behaviors can occur as a consequence of one set of inputs. When Frama-C is invoked and given the annotated code of a function, the contract can be verified against the implementation with *wp*.

Another ACSL concept, i.e., *ghost code*, enables the use of C code within annotations, which is helpful when specifying more complex behaviors, e.g., loops. Ghost code makes implicit information explicitly visible and addressable in function contracts without affecting the behavior of the original source code under analysis [11]. In this work, *ghost code* is applied to transfer stateful information along a call hierarchy (in Listing 6) and to abstract from low-level OS mechanisms (in Listing 9).

A *memory model* is employed to map the analyzed high-level memory concepts of types and pointers to a mathematical representation. An example is *wp*'s default *typed* memory model. To aid the abstraction, memory locations and pointers can be annotated with several terms and predicates predefined in ACSL. A valid pointer `p` that can be safely dereferenced, is declared by `\valid(p)`. All the modified memory locations are listed in the `assigns` clause within the contract. A function or its behavior that has no side effects and assigns no nonlocal memory can be annotated with `assigns \nothing` [10]. In addition, the ACSL annotations are preprocessed and integrated into the abstract syntax tree (AST), which is built using a modified form of the C intermediate language (CIL). It specifies the transformation of C programs into a reduced subset of C, which abstracts from low-level language concepts, supports compiler-specific extensions, and facilitates automatic analyzes. Furthermore, the program is type-checked during the transformation. Afterward, several syntactical transformations are performed, e.g., a unified representation for loops and conditional branches and the removal of "syntactic sugar" like the convenience operator for dereferencing pointers [21], [22], [33].

Analyzes with *wp* are launched either for a complete function contract or for its properties individually. For the selected properties, proof obligations are generated in a *wp*-own syntax that describe the goals to be proven based on the first order logic representation of the analyzed code and its annotations. These obligations are simplified by the builtin *Qed* engine, by either fully resolving them or adding further conditions facilitating the proof [20]. If they are not resolvable by Qed, obligations are forwarded to an automatic SMT prover in the form of a *Why3* script [12]. If existing provers are not sufficient, interactive proof assistants such as Coq [3] can also be utilized to complete the verification [9].

IV. Verification Framework

In this section, the framework for formally verifying the resource synchronization protocols is presented. We first show the common assumption that is made for our framework. Afterward, the workflow of deductive verification used in this work is illustrated. At the end, the necessary preprocessing for the RTOS source code is discussed.

A. Common Assumption

In this work, only the implementation of a protocol is verified. The proposed framework is applied to verify the correctness of the implementation for all the specified properties from a given protocol, i.e., function contracts. Any other components that are not specified in the protocol definition are assumed to be functionally correct, which is shown in Fig. 1. More precisely, the proposed framework assumes that an implemented resource synchronization protocol is based on a correct underlying OS.

In order to verify the implementation layer separately from its underlying layers with deductive program verification, several abstractions have to be applied. First of all, the verification scope does not include the basic locking and scheduling operations, e.g., mutexes, queues, and threads. The mutually exclusive execution of critical sections is considered as a part of the dependencies which are assumed to be correctly implemented. Furthermore, no notion of time is considered. Due

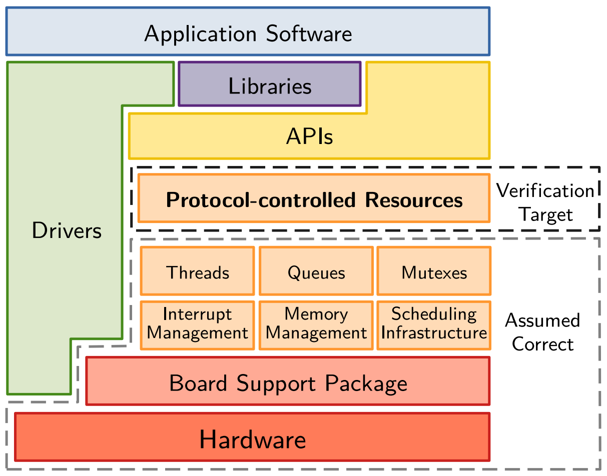

Fig. 1. Abstracted layers of the verification concept within an RTOS.

to the verification perspective and the assumptions on underlying OS concepts, temporal properties are not necessary to verify the protocol specifications. Thus, if all the determined properties of a protocol have been verified to be implemented correctly, the protocol in the OS is formally verified.

B. Workflow of Deductive Verification

Verifying the implementation can be clarified as *neither* verifying its compiled, i.e., compiler- and architecture-specific results, *nor* its behavior in execution. Instead, the source code of the corresponding implementation should be verified with formal specifications of all required properties of the resource synchronization protocol. This can be achieved by deductive program verification, which proves whether a program fulfills a set of postconditions when assuming a set of preconditions.

When the deductive verification approach is applied, the implementation under verification can be written in high-level programming languages. Additionally, to allow for the separation of protocol-specific code and relied-upon OS primitives, the analysis should be performed in a modular way. That is, a set of conditions for a function body should be verified based on its statements, and for further called functions, based on their formal specification only. These called functions either need to be verified if they are specified by the protocol or can be assumed to be correct if they are provided by the RTOS. Hence, the verification can be conducted within a certain level. The protocol-specific code that is to be verified and its dependencies have to be distinguished clearly. Precisely, the workflow of our framework can be described as follows.

1) Identify the subset of the targeted OS's source code that represents the protocol implementation.
2) Identify the resource synchronization protocol's properties and rules, e.g., the request and release of resources.
3) Design specifications consisting of Hoare-triples for the behavior of all utilized OS primitives.
4) Design specifications consisting of Hoare-triples for the protocol implementation based on the formal specification of the protocol.

5) Verify the specification of the implementation against its source code with deductive program verification, aided by appropriate software.

C. Preprocessing the Source Code

To apply Frama-C for verification, the proposed framework only needs to utilize the cross-compilation toolchain without building or executing the OS code. However, the source code of the targeted RTOS needs to be preprocessed to resolve the inclusion dependencies and gain meta information from macros and customized data types [21]. In order to describe the customized data types in the memory model, information on the bit width is also required. Furthermore, some header files are architecture-dependent. These headers may come with the cross-compilation toolchain or be generated during the source configuration.

To analyze the implementations in RTEMS, a separate source configuration is generated for uni- and a multiprocessor configurations, respectively. To avoid compatibility problems, 32-bit PowerPC is chosen as the target architecture for the toolchain, which comes with SMP support in RTEMS and is supported by Frama-C as well [21]. Building the toolchain yields the required header `stddef.h`. Please note that the term *thread* used in RTEMS is the same as *task* in this work and also in the literature of real-time systems.

V. VERIFICATION OF ICPP IN RTEMS

In this section, we adopt the proposed verification framework to verify the ICPP officially implemented in RTEMS [14], which is a well-known synchronization protocol for uniprocessor real-time systems [15]. It is commonly considered as an advanced variant of the PCP [37], as it has the same upper bound on the blocking time but less context switches. However, what is the standard implementation of ICPP has never been discussed. Any mismatch between the implementation and the formally proved properties can potentially lead to an error, e.g., deadlock.

Throughout our verification framework, we find out that the current implementation is in fact *not deadlock free*. To reach this serious conclusion, in the following we present how we declare the function contracts for ICPP to employ our verification framework, and give a concrete example to elaborate how the deadlock can happen under the current implementation. The verified functions are listed in Table I, where the `_CORE_ceiling_mutex` is the common prefix of all function names in the table. All the required properties of ICPP are as follows.

1) For a resource R_j, the priority ceiling is defined as $\Pi(R_j) = \max\{\pi(\tau_i)|\tau_i \text{ requests } R_j\}$, where $\pi(\tau_i)$ is the priority of task τ_i.
2) The set of the resources' priority ceilings that a task τ_i holds at time t is denoted as $C_{\tau_i,t} = \{\Pi(R_j)|\tau_i \text{ holds } R_j \text{ at time } t\}$.
3) At any time t, a task runs at the highest priority among its base priority and the priority ceilings of its held resources: $\pi(\tau_i, t) = \max\{\pi(\tau_i),\ C_{\tau_i,t}\}$.

TABLE I
FUNCTIONS FOR ACQUIRING AND RELEASING A RESOURCE UNDER ICPP

Protocol Function	Purpose
`_CORE_ceiling_mutex[...]`	
`_Seize`	Acquire an available or self-locked resource
$\hookrightarrow$ `_Set_owner`	Check and inherit resource ceiling, set resource owner
`_Surrender`	Release a locked resource

4) Whenever a task τ_i requests a resource R_j at time t, it is granted access and it immediately inherits R_j's priority ceiling: $C_{\tau_i,t} = C_{\tau_i,t-1} \cup \{\Pi(R_j)\}$. Task τ_i executes its critical section with the priority following rule 3.
5) When task τ_i releases a resource R_j at time t, its priority ceiling is revoked from the set, i.e., $C_{\tau,t} = C_{\tau,t-1} \setminus \{\Pi(R_j)\}$. Afterward, task τ_i is executed with its original priority if there is no following critical section, or it is executed for its next critical section with the priority derived by following rule 3.

In the ICPP implementation of RTEMS, after a task successfully locks the semaphore,[1] then the priority of the task is elevated to the ceiling priority if the original priority is lower than the ceiling priority. When a task or thread waits on a semaphore, it is added into a data structure, named as `thread_queue`. The priority queuing discipline just orders the threads according to their current priority and in FIFO order in case of equal priorities.

A. Preprocessing

Before the verification, we decouple the implementation of ICPP into two parts: 1) the protocol-specific parts that will be analyzed and 2) the employed nonspecific OS functionalities. To lock and unlock a resource, the RTEMS Classic API exposes the functions `rtems_semaphore_obtain` and `rtems_semaphore_release`. These functions lock an actual semaphore object from the passed system-wide ID and perform the demanded actions depending on its type. For a semaphore controlled by ICPP, the corresponding functions `_CORE_ceiling_mutex_Seize` and `_CORE_ceiling_mutex_Surrender` are called, where mutex locks are applied as binary semaphores to protect shared resources. Besides, another protocol-specific function is `_CORE_ceiling_mutex_Set_owner`. The remaining functions are lower-level primitives, which provide operations to update a thread's priority, achieve basic mutual exclusion for data consistency or access the underlying nonprotocol *Core Mutexes* and queues, are assumed to be implemented correctly.

Since the implementation of a protocol may be spread across the source base, two header files are created to bundle these functions' specifications: 1) `fc_common_stubs.h` is used for all implemented protocols and 2) `fc_icpp_stubs.h` contains the ICPP-specific stub definitions.

[1]The locking protocols are originally for mutex, but they are realized by binary semaphores in RTEMS, which are technically as mutex locks. Here, we stick to the terminology of locking protocol to "lock" a semaphore.

```
//@ assigns \nothing;
RTEMS_INLINE_ROUTINE void _CORE_mutex_Acquire_critical(
  CORE_mutex_Control *the_mutex,
  Thread_queue_Context *queue_context );
```

Listing 3. Bypassing of a system utility function.

```
/*@
  requires \valid(the_thread) && \valid(priority_node);
  assigns *the_thread, g_thread_inherited, g_prio_node;
  ensures g_thread_inherited == the_thread && g_prio_node ==
     priority_node;
  behavior inherit_higher:
  assumes priority_node−>priority < Current_Priority(the_thread);
  ensures Current_Priority(the_thread) == priority_node−>priority;
  behavior inherit_lower_or_equal:
  assumes priority_node−>priority >= Current_Priority(the_thread);
  ensures Current_Priority(the_thread) ==
  \old(Current_Priority(the_thread));
  disjoint behaviors;
  complete behaviors;
*/
void _Thread_Priority_add(
  Thread_Control *the_thread,
  Priority_Node *priority_node,
  Thread_queue_Context *queue_context );
```

Listing 4. System utility function adds a priority node to a thread.

B. Abstractions and Function Contracts

The OS utilities are treated in two different ways when they are annotated to declare their (intended) behavior in the analysis. On the one hand, functions that have no effect in the analyzed situation, are "bypassed". That is, the annotations do not declare their behavior, but assert their invocation has no side effects and can be ignored during verification. Listing 3 shows an example for bypassing calls for basic locking pairs, where the interrupt has already been disabled when the function is called in a uniprocessor system. On the other hand, OS functions that perform actions that are not considered as a part of the protocol analysis but are critical to the ICPP implementation, have to be annotated with a description of their intended (and considered correct) behavior. The example in Listing 4 shows a contract that ensures the thread's priority either remains the same or corresponds to the passed priority node after its execution.

C. Contracts for ICPP-Seize

Once all necessary OS functions have been provided with contracts, the actual behaviors of the protocol operations *seize* and *surrender* can be specified. The following two properties are verified for seize in a pure ICPP.

1) A task requesting a resource is granted the resource.
2) After a resource is granted, the task runs on the highest ceiling priority of all currently held resources.

The implementation in RTEMS considers and checks more possible cases, which are not formally described by the protocol specifications. Overall, these additional scenarios lead to further properties.

1) Acquiring a resource fails, if its priority ceiling is lower than the acquiring thread's base priority.
2) Resources may be acquired again by the holding thread before release. The level of self-nested access is tracked.

```
1  /*@
2    requires \valid(the_mutex) && \valid(executing);
3    requires \separated(the_mutex, executing);
4    requires Mutex_Owner(the_mutex) == NULL ||
5    Mutex_Owner(the_mutex) == executing;
6    behavior seize_ceiling_violation:
7      assumes Mutex_Owner(the_mutex) == NULL && Base_Priority(
          executing) < Mutex_Priority(the_mutex);
8      ensures \result == STATUS_MUTEX_CEILING_VIOLATED;
9    behavior seize_successful:
10     assumes Mutex_Owner(the_mutex) == NULL && Base_Priority(
          executing) >= Mutex_Priority(the_mutex);
11     ensures PriorityInherited(executing, Mutex_Priority(the_mutex));
12     ensures Current_Priority(executing) <= Mutex_Priority(the_mutex);
13     ensures Mutex_Owner(the_mutex) == executing;
14     ensures \result == STATUS_SUCCESSFUL;
15   behavior seize_nested:
16     assumes Mutex_Owner(the_mutex) == executing;
17     assumes nested == _CORE_recursive_mutex_Seize_nested;
18     assigns the_mutex−>Recursive.nest_level;
19     ensures Nest_Level(the_mutex) == \old(Nest_Level(the_mutex)) + 1;
20     ensures \result == STATUS_SUCCESSFUL;
21   disjoint behaviors;
22 */
23   RTEMS_INLINE_ROUTINE Status_Control
       _CORE_ceiling_mutex_Seize(
24     CORE_ceiling_mutex_Control *the_mutex,
25     Thread_Control *executing,
26     bool wait,
27     Status_Control ( *nested )( CORE_recursive_mutex_Control * ),
28     Thread_queue_Context *queue_context
29 ) { /*...*/
30   //@ calls _CORE_recursive_mutex_Seize_nested;
31   status = ( *nested )( &the_mutex−>Recursive );
32   /*...*/ }
```

Listing 5. Contract declaring the ICPP functionality for the seize operation.

3) Acquiring a locked resource enqueues the thread into the resource's priority-based waiting queue. Such a request operation can be either successful by obtaining the resource eventually or failed if the request times out.

The check of the acquiring task's base priority is a legitimately safe measure to compensate incorrectly priority ceilings setting or unallowed resource accesses. Locking an already locked resource does not affect the ICPP. The last case happens only if a task suspends while holding a resource. This behavior is not considered in the definition of ICPP and would break the property. Toward this, the precondition is necessary that the requested resource is either available or locked by the current requesting task (Listing 5, ll. 4 and 5), which matches the formal property of ICPP, i.e., once a task starts its execution, all required resources must be available [14]. Therefore, the third case in the list is excluded in the analysis, which makes annotations for the responsible function `_CORE_mutex_Seize_slow` unnecessary.

These collected properties can be formulated as a function contract for the function `_CORE_ceiling_mutex_Seize`, shown in Listing 5. The preconditions in lines 2 and 3 require that the pointers to the mutex and the executing thread are valid, i.e., dereferenceable, and their memory regions do not overlapped. The precondition in lines 4 and 5 expresses the invariant for a seize operation under ICPP, that the requested resource is free, and adds the situation that the shared resource has been acquired by the requesting task already. The default behavior of a successful acquisition, `behavior seize_successful`, ensures that the

```
/*@ ghost // variables declared in coremuteximpl.h
  extern Thread_Control *g_thread_inherited;
  extern Thread_Control *g_thread_revoked;
  extern Priority_Node *g_prio_node;
  extern bool prioritiesUpdated; */
/*@ predicate PriorityInherited(Thread_Control *t, Priority_Control p) =
  t == g_thread_inherited && p == g_prio_node−>priority &&
      prioritiesUpdated;
*/
```

Listing 6. Predicate `PriorityInherited` checks priority inheritance.

```
/*@
  requires \valid(the_mutex) && \valid(owner);
  requires \separated(the_mutex, owner);
  behavior set_owner_ceiling_violation:
    assumes Base_Priority(owner) < Mutex_Priority(the_mutex);
    ensures \result == STATUS_MUTEX_CEILING_VIOLATED;
  behavior set_owner_successful:
    assumes Base_Priority(owner) >= Mutex_Priority(the_mutex);
    assigns *owner, *the_mutex, prioritiesUpdated, g_thread_inherited,
       g_prio_node;
    ensures Current_Priority(owner) <= Mutex_Priority(the_mutex);
    ensures PriorityInherited(owner, Mutex_Priority(the_mutex));
    ensures Mutex_Owner(the_mutex) == owner;
    ensures \result == STATUS_SUCCESSFUL;
  disjoint behaviors;
  complete behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control
   _CORE_ceiling_mutex_Set_owner(
  CORE_ceiling_mutex_Control *the_mutex,
  Thread_Control *owner,
  Thread_queue_Context *queue_context ){/*...*/}
```

Listing 7. Contract for the `_Set_owner` function.

resource is locked by the requesting thread and the resource's ceiling priority is inherited. To ensure that, the inheritance is performed from the acquired resource to the requesting thread, the predicate `PriorityInherited` (Listing 6) is introduced. It checks if the passed thread and priority are the same as those set in the contract of `_Thread_Priority_add` (Listing 4) and whether the priority of the thread is updated after the change of the priority aggregation. The inheritance does not necessarily lead to a priority raise, since the thread may already hold a resource with a higher ceiling.

Within the seize function, the second relevant protocol-specific function from Table I is the `_Set_owner` function in Listing 7. It performs the check of the resource ceiling. If valid, the executing thread inherits the acquired resource's priority ceiling and is set as the resource owner. If the resource ceiling is violated due to the thread's priority, the operation fails. The conditions for the ceiling and the ceiling priority inheritance are known from the invoking seize function.

D. Contracts for ICPP-Surrender

The contract for the ICPP surrender function, i.e., `_CORE_ceiling_mutex_Surrender`, can be designed in a similar sense. Since the ICPP does not allow threads to be enqueued and waiting for the shared resource, the contract is constructed with a precondition that the resource's queue has to be empty. The revocation of the formerly inherited priority is guaranteed by another predicate, i.e., `PriorityRevoked`. The thread's dynamic priority after surrendering the resource is either lower than before or remains the same if another resource with the same ceiling is still held.

```
if (
  owner−>Real_priority.priority
  < the_mutex−>Priority_ceiling.priority
) {
  _Thread_Wait_release_default_critical( owner, &lock_context );
  _CORE_mutex_Release(&the_mutex−>Recursive.Mutex,
     queue_context);
  return STATUS_MUTEX_CEILING_VIOLATED;
}
//@ assert Base_Priority(owner) >= Mutex_Priority(the_mutex);
```

Listing 8. Proposed correction for the priority ceiling check.

E. Verification and Mismatch

Due to the modular analysis, the seize function can be analyzed without inspecting the code of the called `_CORE_ceiling_mutex_Set_owner`. Instead, only its contract is used. Once the function under analysis fulfills that contract's preconditions, its postconditions are assumed to be fulfilled. This analysis successfully proves all stated properties. However, when attempting to verify the called set owner function, the verification fails to prove the ceiling check and parts of the successful acquisition. The reason for the incapability to fulfill the conditions can be tracked down with further annotations.

After a successful ceiling check, the task's base priority is assumed to be lower or equal to the resource's ceiling. However, this assertion cannot be verified. We note that the resource's ceiling is not checked against the thread's base priority, but against its current dynamic priority derived from the task's priority aggregation. However, a resource's ceiling is required to be set as the highest base priority of all tasks that are requesting it. This mismatch may lead to a deadlock by erroneously denying legitimate nested resource access if resources are requested with descending order of priority ceilings. We give an example to elaborate such a case.

Consider two tasks τ_1 and τ_2 with $\pi(\tau_1) > \pi(\tau_2)$ and two resources R_1 (used by both tasks) and R_2 (used by τ_2 only) with $\Pi(R_1) = \pi(\tau_1)$ and $\Pi(R_2) = \pi(\tau_2)$. If τ_2 acquires R_1, it inherits its priority ceiling and executes with the promoted dynamic priority $\pi(\tau_1)$. If it requests the second resource R_2, its dynamic priority is higher than $\Pi(R_2)$, which leads to a denial of the resource access by the implemented ceiling check. The consequence of this is a *deadlock*, i.e., τ_2 holds R_1 but cannot successfully lock semaphore R_2 due to the implemented ceiling check, whilst τ_1 cannot enter the critical section guarded by R_1. Such execution behavior with a deadlock can also be demonstrated by a running example in RTEMS, which will be released on Github. An acquisition in the opposite order would be accepted.

To correct the mismatch, an adaption to the priority ceiling check is proposed in Listing 8 for `coremuteximpl.h`. Instead of checking the potentially already increased dynamic priority, it uses the thread's base priority for the comparison to the newly requested resource's priority ceiling. After applying the correction, all stated properties are successfully verified.

TABLE II
FUNCTIONS FOR ACQUIRING AND RELEASING A RESOURCE UNDER MRSP

Protocol Function	**Purpose**
`_MRSP[...]`	
`_Seize`	Acquire an available or wait for a locked resource
↪ `_Claim_ownership`	Performed if the resource is free
↪ `_Wait_for_ownership`	Performed if the resource is used on another processor
↪ `_Raise_priority`	Always performed to run at the resource's local ceiling
`_Surrender`	Release a locked resource and pass it to the first (if any) waiting task

VI. VERIFICATION OF MRSP IN RTEMS

In this section, we verify the MrsP [13] officially implemented in RTEMS, which is designed for semi-partitioned FP task systems on multiprocessors. We adopt our verification framework to ensure whether the corresponding implementation derives the specified properties of the MrsP. One highlight of the MrsP is the help mechanism that employs a spin-waiting task to progress the execution of the current blocked task which holds the resource. However, a seize operation can be performed while being scheduled in the presence of the help mechanism. This requires the priority ceiling of the seized resource to be determined with a caution, which is of key interest in this work.

The protocol functions that are going to be verified are listed in Table II, where _MRSP is the common prefix of all function names in the table. From the verification perspective, similar preprocessing in Section V-A is necessarily operated for the implementation of MrsP as well. The help mechanism can be assumed to be implemented correctly as other OS utilities, as long as dynamic priorities and ceilings are managed correctly. In addition, the verification is based on one arbitrary thread that performs the analyzed operation. Any other threads which might interact with it are assumed to behave correctly. A successful verification implies that this assumption holds as well. The properties of the original MrsP are as follows.

1) Each task τ_i is assigned to a specified processor P_m, and critical sections are executed locally, unless the *help mechanism* is applied.
2) Each resource R_j has one local priority ceiling for each processor P_m, which is defined by the highest priority of every task assigned to P_m that requests the resource: $\Pi(R_j, P_m) = \max\{\pi(\tau_i) \mid \tau_i \text{ requests } R_j \text{ on } P_m\}$.
3) For local resources that are not shared between processors, the ICPP rules are applied.
4) For global resources, the ICPP inheritance mechanism is applied with their local priority ceilings. If the requested global resource is not available, the requesting tasks spin-wait on their own processor in a FIFO order.
5) Help mechanism: a spin-waiting task for accessing to a resource must be able to *help* (by offering its computation time to) the current owner of the resource in case the owner is preempted within the critical section.

```
// variable defined in mrspimpl.h
//@ ghost extern Scheduler_Node *g_homenode;
/*@ requires \valid(the_thread);
  assigns \nothing;
  ensures \result == g_homenode; */
RTEMS_INLINE_ROUTINE Scheduler_Node *
   _Thread_Scheduler_get_home_node(const Thread_Control *the_thread
   );
```

Listing 9. Abstraction of the function that returns a thread's home node.

In fact, the help mechanism in the original design of the MrsP by Burns and Wellings [13] can cause additional local blocking, since threads are allowed to acquire priority-promoting resources while being helped on other processors, which may preempt threads dispatched on their home processors. Garrido *et al.* [27] suggested to resolve this issue by postponing the effect of inherited priorities to the time when the thread returns to its home processor. The verified implementation coincidentally realizes the same concept by dispatching idle threads to run subsidiary for threads that migrate to seek help by Catellani *et al.* [16].

A. Abstractions and Function Contracts

The multiprocessor setup requires further abstractions and adaptions. While some OS utilities' stub contracts designed for the verification of ICPP can be reused, the others need to be wrapped with a new contract. For example, the function in Listing 9 retrieves a thread's *home node*, i.e., the scheduler node for its original processor. However, it is retrieved as a chain element via several nested function calls and then extracted by a macro. This macro is expanded over multiple definitions and is eventually based on a compiler-specific offset function, which is not able to be formulated in ACSL contracts or logic functions.

Since the derivation reaches deeply into the OS specific functions, it becomes a target for abstraction. Instead of tracing the complete call and macro hierarchy, we declare a global ghost pointer `g_homenode` of the type `Scheduler_Node` to represent the executing thread's home scheduler node in the context of the verification. The `getter` function is specified by an ACSL contract to return a reference to that scheduler node in Listing 9. The ghost object is then said to be valid by the preconditions of the verified functions, which ensures the validity of dereferencing and access to its fields. Therefore, we consider the ghost object is equivalent to a scheduler node retrieved by the original utility function.

We also need to abstract the local resource priority ceilings for each processor. A task has to raise its priority to the local priority ceiling of the resource that is requested. From the verification perspective, the task's assigned processor may be arbitrary, but fixed, and can be modeled by another ghost variable `const int g_core`. The maximum number of processors configured in the architecture-specific `cpu.h` files is 32, which can be formulated as a constraint for the variable `g_core`. It also defines the amount of valid entries for a resource's local ceilings. The detailed function contract is omitted here due to the space limitations.

The inheritance and revocation of priorities are modeled by the predicates `PriorityInherited` and `PriorityRevoked` similarly to the verification of ICPP.

B. Contracts for MrsP-Seize

When designing the contracts for the seize operation and its corresponding functions, a ceiling check similar to the function, i.e., `_CORE_ceiling_mutex_Set_owner` in ICPP is detected. Inside `_MRSP_Raise_priority`, the priority of the executing thread is checked against the local ceiling of the requested resource. The comparison is performed with the current scheduler node's dynamic priority rather than with the thread's base priority. As a result, the current implementation of the seize operation of MrsP in RTEMS does not allow an arbitrary sequence of resource requests if they are not properly nested. However, the implementation is valid only under one assumption: a thread acquiring nested resources always requests them in a nondescending order of priority ceilings. The assumption is translated to a precondition in the successful behaviors of the affected functions.

A resource that is additionally acquired while being helped by a waiting thread does not necessarily have a priority ceiling for the foreign processor to which the thread has migrated. Instead, the migrated thread always inherits the resource's local ceiling stored for its home processor and does not affect the priority of the helping thread. This feature indicates that the ceiling check correction in Section V-E could be applied for multiprocessor systems as well. As long as the thread still holds the resource it is helped with, it stays in the migrated-to processor and runs at a legitimate priority. The inheritance of a new ceiling becomes effective as soon as the thread returns back to its home scheduler.

C. Contracts for MrsP-Surrender

The surrender operation for MrsP has to be handled carefully due to possible waiting threads. The contract is shown in Listing 10, where the possible waiting tasks can revoke the surrendered resource's priority ceiling. Threads waiting for the resource spin in the corresponding FIFO queue. Therefore, based on the contents of that queue, we can distinguish the behavior with the assistance of the predicate `MrsPThreadsWaiting` in Listing 11. In case there is no waiting thread, the resource owner (which corresponds to the queue owner) is simply reset to `NULL`. On the other hand, if the waiting queue is not empty, the first thread is set to be the succeeding owner. These operations are ensured by the stub contract for the queue's surrender function `_Thread_queue_Surrender_sticky` in Listing 12. This function ensures that the ownership is passed to the next thread and the affected tasks' priorities are updated. The thread as the new resource owner can be abstracted by the ghost variable `g_new_owner`. In both cases, the surrendering thread loses the inherited priority.

The actual transfer of the ownership happens in the counterpart during the waiting thread's seize operation, by the queue function `_Thread_queue_Enqueue_sticky`. When the

```
/*@
  requires \valid(mrsp) && \valid(&mrsp->Wait_queue.Queue) &&
  \valid(executing) && \valid(g_homenode);
  behavior surrender_no_successor:
    assumes MrsP_Owner(mrsp) == executing;
    assumes ! MrsPThreadsWaiting(mrsp);
    ensures MrsP_Owner(mrsp) == NULL;
    ensures PriorityRevoked(executing, MrsP_Ceiling(mrsp));
    ensures Executing_Priority >= \old(Executing_Priority);
    ensures \result == STATUS_SUCCESSFUL;
  behavior surrender_successor:
    assumes MrsP_Owner(mrsp) == executing;
    assumes MrsPThreadsWaiting(mrsp);
    ensures PriorityRevoked(executing, MrsP_Ceiling(mrsp));
    ensures Executing_Priority >= \old(Executing_Priority);
    ensures MrsP_Owner(mrsp) == g_new_owner;
    ensures \result == STATUS_SUCCESSFUL;
  behavior surrender_fail:
    assumes MrsP_Owner(mrsp) != executing;
    ensures \result == STATUS_NOT_OWNER;
  disjoint behaviors;
  complete behaviors;
*/
RTEMS_INLINE_ROUTINE Status_Control _MRSP_Surrender(
  MRSP_Control *mrsp,
  Thread_Control *executing,
  Thread_queue_Context *queue_context ){/*...*/}
```

Listing 10. Function contract of the MrsP surrender operation.

```
/*@ predicate MrsPThreadsWaiting(MRSP_Control *m) =
  m->Wait_queue.Queue.heads != NULL;
*/
```

Listing 11. Helper predicate to determine if a resource wait queue is empty.

```
/*@
  requires \valid(queue);
  requires queue->owner == NULL;
  assigns queue->owner, prioritiesUpdated;
  ensures queue->owner == g_new_owner;
  ensures prioritiesUpdated;
*/
void _Thread_queue_Surrender_sticky(
  Thread_queue_Queue *queue,
  Thread_queue_Heads *heads,
  Thread_Control *previous_owner,
  Thread_queue_Context *queue_context,
  const Thread_queue_Operations *operations );
```

Listing 12. Stub contract for the surrender operation on a queue for MrsP.

function is called successfully, the calling thread is guaranteed to receive the ownership of the queue. Furthermore, the waiting thread is expected to have raised its priority to the resource's local priority ceiling as expressed by the annotations of the seize operation. Therefore, the surrendering thread does not have to take care of priority manipulations for other tasks.

D. Verification With Frama-C

As explained earlier, the official implementation of the seize operation of MrsP in RTEMS does not allow an arbitrary sequence of resource requests if they are not properly nested. We have to additionally introduce an assumption, i.e., a thread acquiring nested resources always requests them in a nondescending order of priority ceilings, together with the derived function contracts. After applying the remedy of ICPP and the introduced assumption, all the implemented functions

TABLE III
DERIVED PROOF GOALS AND REQUIRED TIME OF THE VERIFICATION OF ICPP AND MRSP

	Proof Goals		Time
	Qed	Alt-Ergo	
ICPP Functions	51	44	$\approx 8,6s$
MrsP Functions	75	27	$\approx 7s$

are successfully verified with Frama-C and *wp*, i.e., satisfying the original definition of MrsP. We also ensure that the help mechanism conforms to the suggestions proposed by Garrido *et al.* [27]. In case the implementation of the ceiling check and the priority retrieval are changed, the annotations can be adapted and the verification can be reattempted.

VII. OVERHEAD AND DISCUSSION

In this section, we report the required overhead of verifying ICPP and MrsP supported in RTEMS. Afterward, we discuss the challenges of verifying synchronization protocols and extensibility of the proposed framework.

A. Annotation and Runtime

To verify the protocol-specific functions, we annotate every encountered function, together with appropriate abstractions from the OS's details. The annotations for the protocols and OS utilities comprise 318 lines in total which subdivide into 123 lines for ICPP, 186 lines for MrsP, and 9 lines for commonly used annotations. These numbers only count ACSL annotations, excluding copied function declarations, blank lines, and lines containing only opening/closing symbols for comments. The corresponding source code can be reviewed in [4]. Please note that the verification framework does not build or execute the annotated source code, so the annotation is only for the verification purpose.

To determine the required time for performing the verification, we executed Frama-C without GUI and verify the protocol functions by passing them via the argument `-wp-fct f1,f2,..., fn` to Frama-C. The process was executed single-threaded on an Intel Core i5-4200U CPU with 12 GB of main memory. The computation time was captured by the `time` command and includes preprocessing, transformation, and normalization of the protocol implementation, the generation and simplification of proof obligations as well as the delegation of selected proofs to Alt-Ergo [1]. The results for derived proof goals and required time of the verification of ICPP and MrsP are listed in Table III.

B. Discussion

Although there are corresponding functions for the acquire and release operations in both ICPP and MrsP protocols with `..._Seize` and `..._Surrender`, the distinction of protocol-specific functions is still challenging. In RTEMS, the implementations of the protocol rules are spread over various subfunctions. Furthermore, the implementation includes several additional checks and actions, which are not formally described by the protocol specifications.

For ICPP, several additional check mechanisms are added.

1) Check the correctness of setting the resource's priority ceiling.
2) Check if the task reclaims an already locked resource or claims a locked resource by enqueueing.
3) Check if the task is allowed to access the resource.
4) The resource owner actively switches its state to blocked, i.e., self-suspending.

For MrsP, the priority ceiling check is included as well, along with an extension which runs idle tasks to substitute helped tasks during their migration phase to improve nested resource access.

These rules above have to be split across the function contracts, where a called function has to be one part of the caller's function contract. The call for one function can generate a call chain of protocol-specific functions. A leaf function's preconditions have to be ensured by the calling functions. In turn, its ensured postconditions can be relevant to the root function along with its contract.

Our case studies show that a protocol is not necessarily implemented exactly as specified. In practice, the implementation has to cover a broad variety of possible configurations, e.g., a task requests a self-locked resource or the resource owner is self-suspending. The approach to handle such difference in this work is to require ICPP's invariant as a precondition. Then the protocol-conforming subset of the actual implementation can be verified efficiently.

With the introduced techniques, further properties could be verified. For example, the basic locking which was not considered as part of the verification could be included. A possible approach is to introduce a boolean ghost status variable for each locking level, e.g., for the thread-, mutex-, and MrsP-queues. Since the analyzed functions are called from an API function, the state of the locks at the time of the call must be included in the preconditions of the called functions. The contracts of the locking primitives that are currently bypassed could be changed to ensure the correct state of their affected locks in the postcondition. The contracts of the protocol functions could then express the properties of the locks by referring to the ghost variables' states. To complete the specification of memory assignments, some top-level functions, e.g., `_MRSP_Seize` and `_MRSP_Surrender` would have to be complemented with `assigns` clauses in order to make sure that they have no unspecified side effects. Such annotations become important if these functions are included in a possible verification of the invoking API functions.

The proposed verification framework can also be applied on other OSes, with the assumption that all the low-layer functions are implemented correctly in the targeted OS, e.g., mutexes, queues, and threads. For each targeted OS, the definitions of the used helper predicates and logic functions have to be adapted. Once the necessary OS utilities are abstracted, these basis can be used to verify the implementations of multiple protocols. For each implemented protocol, the properties derived from the formal definition for the seize and the surrender operation should be portable to the targeted

OS with reasonable effort. In the end, the function contracts for the detailed implementation of a dedicated resource synchronization protocol can be designed and verified.

VIII. Related Work

In this section, related work on formal verification (in general, for RTOSes, and for specific programs) is presented to position the contribution of this work.

A. Formal Verification

Frama-C has been widely used for various applications. For example, Efremov *et al.* [23] proposed a method for the deductive verification of Linux kernel functions, where a new plugin was developed due to the incompatibilities of Frama-C and certain kernel constructs. A similar approach to deductive verification, the verifier for concurrent C (VCC), was developed by Cohen *et al.* [19]. The program correctness is verified to hold for every possible concurrent execution of threads during runtime, which is enabled by tracking the ownership of (nonvolatile) data. VCC has been applied to partially verify Microsoft's Hyper-V Hypervisor [32] and a small exemplarily implemented Hypervisor [5]. Deductive verification is also applied for further languages aside from C. An example is the *Prusti* project [7], which enables deductive program verification for Rust by function contracts similar to those adopted in this article.

B. Formal Verification of Operating Systems

Several approaches have been developed and evaluated that only focus on the formal verification of OSs. One concept aims at designing and implementing OS kernels with complete formal verification from the beginning, instead of attempting to verify existing kernels. A concrete example is *seL4*, which is presented by Klein *et al.* [30]. The microkernel is verified via refinement steps from the abstract specification represented by a Haskell prototype over the executable specification in Isabelle/HOL to a manually optimized C version. Every layer below the verified source code of the microkernel, from the compiler to the hardware, is assumed correct and not target of the verification. Gu *et al.* [28] presented an architecture for concurrent OS kernels consisting of multiple layers. The code of each system layer is verified with Coq. As a demonstration of the architecture, the kernel *mC2* was developed for multiprocessor x86 computers, supporting fine-grained locking, threads with suspension, and serving as a hypervisor.

An approach that targets RTEMS has been presented by Gadia *et al.* [25]. In order to verify the implementation of the PIP with a software model checker, it was remodeled in Java along with the relevant associated scheduling mechanisms. PIP- and race-condition-related safety properties were included as assertions and the resulting model was investigated with *Java Pathfinder*. During the evaluation, an implementation error related to nested resource sharing was confirmed and fixed. Additionally, Almatary *et al.* [6] proposed an approach to reduce the kernel calls when implementing ICPP in POSIX, where the implementation is verified by using model checking.

Recently, Nicole *et al.* [34] has proposed an approach to automatically verify two properties of a given OS: the absence of runtime errors (ARTE) and privilege escalation (APE). The verification target is represented by the binary executable. An abstract interpreter was developed to process the executable and determine all possibly reachable states of registers and memory. Based on this, ARTE and APE can be verified automatically. Compared to verifying source code, this approach is highly specific to the instruction set architecture for which the image was built, and the verification is restricted to critical, but fixed low-level properties.

C. Formal Verification of Real-Time Programs

There are some works focusing on verifying specific real-time programs that utilize locks as well. Chaki *et al.* [17] proposed an approach to verify safety and deadlock freedom of programs with PIP locks. Their approach is based on sequentialization. That is, a periodic program is converted into an equivalent (nondeterministic) sequential program at first. Afterward, a model checker is applied for verifying the correctness. In addition, Suresh *et al.* [39] proposed a technique that can statically detect data races in periodic real-time programs with locks on uni-processor systems. Their approach is based on a small set of rules that exploit the priority, periodicity, locking, and timing information of tasks in the program.

They focused on verifying specific programs that use locks with concrete multiple tasks and resources. However, in this work, we proposed a framework that can be applied to verify whether the properties of the targeted resource synchronization protocol are fulfilled in the corresponding implementation, which is independent from specific use cases.

IX. Conclusion

In real-time systems, various resource synchronization protocols have been proposed since the 1990 s for concurrent tasks that share resources. Most of the studies focus on the worst-case timing analysis in theory. Only a few work discusses the realization details and pitfalls in practice. Although many protocols are nowadays supported in different RTOSes, how to ensure the implemented protocols (e.g., often from many contributors) can always comply the proved properties is a challenging problem to the community.

In this work, we present a pragmatic framework to verify the existing protocol implementations in RTOSes. We propose to specify the intended behaviors of the implementation in the form of function contracts. The deductive verification is applied to verify whether each implemented component matches its formally described properties under the assumption that the underlying primitives are implemented correctly. We propose a verification framework to enable modular formal verification of functional components. The analysis target is defined and isolated from its dependencies, so that the workflow of the proposed framework is conceptually independent to the platform.

The case studies for ICPP and MrsP implemented in RTEMS show the applicability of the proposed verification framework to an actual RTOS. Moreover, the discovery of the mismatches in the RTEMS implementations shows its functionality. After a proposed correction, the implementations of ICPP and MrsP (under one additional assumption) can be successfully verified. With the success of verifying the protocols, we plan to verify other system software in different RTOSes for future work.

References

[1] "An SMT solver for software verification." Feb. 2022. [Online]. Available: https://alt-ergo.ocamlpro.com/

[2] "Frama-C software analyzers." Jan. 2022. [Online]. Available: https://frama-c.com/

[3] "The Coq proof assistant." Jan. 2022. [Online]. Available: https://coq.inria.fr

[4] "The source code of this work." Accessed: Jul. 26, 2022. [Online]. Available: https://github.com/tu-dortmund-ls12-rt/Resource-Synchronization-Protocols-Verification-RTEMS

[5] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova,. "Automated verification of a small hypervisor," in *Verified Software: Theories, Tools, Experiments*. Edinburgh, U.K.: Springer, 2010, pp. 40–54.

[6] H. Almatary, N. C. Audsley, and A. Burns, "Reducing the implementation overheads of IPCP and DFP," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2015, pp. 295–304.

[7] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," in *Proc. Object Oriented Program. Syst. Lang. Appl.*, vol. 3, pp. 1–30, 2019.

[8] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[9] P. Baudin, F. Bobot, L. Correnson, Z. Dargaye, and A. Blanchard. *WP Plug-In Manual, Version 22.0*. (2020). [Online]. Available: https://frama-c.com/download/wp-manual-22.0-Titanium.pdf

[10] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, Version 1.16 for Frama-C 22.0*. (2020). [Online]. Available: https://frama-c.com/download/acsl-implementation-22.0-Titanium.pdf

[11] A. Blanchard. *Introduction to C Program Proof With Frama-C and Its WP Plugin*. (2020). [Online]. Available: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

[12] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd your herd of provers," in *Proc. Boogie 1st Int. Workshop Intermediate Verif. Lang.*, 2011, pp. 53–64.

[13] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol—MrsP," in *Proc. 25th Euromicro Conf. Real-Time Syst. (ECRTS)*, 2013, pp. 282–291.

[14] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages—Ada, Real-Time Java and C/Real-Time POSIX* (International Computer Science Series), 4th ed. Harlow, U.K.: Addison-Wesley, 2009.

[15] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications* (Real-Time Systems Series), 3rd ed. New York, NY, USA: Springer, 2011.

[16] S. Catellani, L. Bonato, S. Huber, and E. Mezzetti, "Challenges in the implementation of MrsP," in *Reliable Software Technologies—Ada-Europe*. Madrid, Spain: Springer, 2015, pp. 179–195.

[17] S. Chaki, A. Gurfinkel, and O. Strichman, "Verifying periodic programs with priority inheritance locks," in *Proc. Formal Methods Comput.-Aided Des. (FMCAD)*, 2013, pp. 137–144.

[18] J.-J. Chen, G. von der Brüggen, J. Shi, and N. Ueter, "Dependency graph approach for multiprocessor real-time synchronization," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, 2018, pp. 434–446.

[19] E. Cohen *et al.*, "VCC: A practical system for verifying concurrent C," in *Proc. Int. Conf. Th. Proving Higher Order Logics*, 2009, pp. 23–42.

[20] L. Correnson, "Qed. Computing what remains to be proved," in *Proc. 6th Int. Symp. NASA Formal Methods*, 2014, pp. 215–229.

[21] L. Correnson *et al. Frama-C User Manual, Version 22.0 (Titanium)*. (2020). [Online]. Available: https://frama-c.com/download/user-manual-22.0-Titanium.pdf

[22] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C—A software analysis perspective," in *Proc. Int. Conf. Softw. Eng. Formal Methods*, 2012, pp. 233–247.

[23] D. Efremov, M. U. Mandrykin, and A. V. Khoroshilov, "Deductive verification of unmodified Linux kernel library functions," in *Proc. Int. Symp. Leverag. Appl. Formal Methods Verif. Validation Verif.*, 2018, pp. 216–234.

[24] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, vol. 19, 1967, pp. 19–32.

[25] S. Gadia, C. Artho, and G. Bloom, "Verifying nested lock priority inheritance in RTEMS with Java pathfinder," in *Proc. Int. Conf. Formal Eng. Methods*, 2016, pp. 417–432.

[26] P.-L. Garoche, *Formal Verification of Control System Software*, 1st ed. Princeton, NJ, USA: Princeton Univ. Press, 2019.

[27] J. Garrido, S. Zhao, A. Burns, and A. Wellings, "Supporting nested resources in MrsP," in *Reliable Software Technologies—Ada-Europe*. Vienna, Austria: Springer, 2017, pp. 73–86.

[28] R. Gu *et al.*, "Certikos: An extensible architecture for building certified concurrent OS kernels," in *Proc. 12th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2016, pp. 653–669.

[29] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[30] G. Klein *et al.*, "seL4: Formal verification of an OS kernel," in *Proc. Symp. Oper. Syst. Principles*, 2009, pp. 207–220.

[31] J.-H. Lee and H.-N. Kim, "Implementing priority inheritance semaphore on uC/OS real-time kernel," in *Proc. 1st Workshop Softw. Technol. Future Embedded Syst.*, 2003, pp. 83–86.

[32] D. Leinenbach and T. Santen, "Verifying the Microsoft hyper-V hypervisor with VCC," in *Proc. 2nd World Congr. Formal Methods*, 2009, pp. 806–809.

[33] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. Int. Conf. Compiler Constr.*, 2002, pp. 213–228.

[34] O. Nicole, M. Lemerre, S. Bardin, and X. Rival, "No crash, no exploit: Automated verification of embedded kernels," in *Proc. Real-Time Embedded Technol. Appl. Symp.*, 2021, pp. 27–39.

[35] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 1990, pp. 116–123.

[36] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proc. Real-Time Syst. Symp.*, 1988, pp. 259–269.

[37] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[38] J. Shi, N. Ueter, G. von der Brüggen, and J.-J. Chen, "Multiprocessor synchronization of periodic real-time tasks using dependency graphs," in *Proc. 25th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, 2019, pp. 279–292.

[39] V. P. Suresh, R. R. Pai, D. D'Souza, M. D'Souza, and S. K. Chakrabarti, "Static race detection for periodic programs," in *Proc. Eur. Symp. Program.*, 2022, pp. 290–316.