Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

BIP-based formal verification of embedded real-time operating systems

Xinxu Chang
Beijing Institute of Computer Technology and Application
Rui Yao
Beijing Institute of Computer Technology and Application
Xinxin Jiang
Beijing Institute of Computer Technology and Application
Yang Zhang
614032144@qq.com

Beijing Institute of Computer Technology and Application
Xiaoyan Zhao
Beijing Institute of Computer Technology and Application
Xia Yang
University of Electronic Science and Technology of China Chengdu

Case Report

Keywords: BIP, embedded real-time operating system, compositional verification, statistical model checking, refinement, observable equivalence, automatic test generation

Posted Date: November 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11671234/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

BIP-based formal verification of embedded real-time operating systems

Abstract

Embedded real-time operating systems (RTOSs) are widely deployed in safety-critical domains where reliability and functional correctness are stringent requirements. This paper reports formal verification of an in-house RTOS and its integration into the development workflow. We use BIP to build requirement-level models of kernel modules, leverage BIP's C++ linking to emulate native RTOS behavior, and automatically convert model-generated traces into test code executable on the target, providing evidence of observable equivalence over the generated traces. The method was applied to six core modules with 23 properties specified; the 15 target properties of the task-management module were verified in full. Four of five safety properties hold under statistical model checking (error probability 0.001, precision 0.05), deadlock freedom is witnessed by a structural progress guard, and all ten functional properties hold under record-level comparison against the kernel. In total 1,638 traces were replayed as kernel tests; model and implementation agree on call order, arguments, return codes and task-state transitions in all scenarios. Results show that module-level models remain statistically analyzable, trace-derived tests localize divergence to one recorded action in one scenario, and verified modules can be reused and regenerated after kernel changes. It provides an incremental, tool-supported path, not exhaustive whole-kernel verification.

Keywords: BIP; embedded real-time operating system; compositional verification; statistical model checking; refinement; observable equivalence; automatic test generation

1 Introduction

Embedded real-time operating systems (RTOSs) are widely deployed in safety-critical domains such as aerospace, defense systems, automotive electronics, industrial control, medical devices, and the Internet of Things (IoT), where reliability and correctness must meet exceptionally high requirements[1,2,3]. As the foundational software platform of the entire system, an embedded RTOS must be reliable and correct, because these properties directly affect the safety and availability of the overall system. Software defects or faults may cause missed task deadlines or performance degradation and, in severe cases, catastrophic consequences. Consequently, embedded RTOSs must rigorously guarantee deterministic task response and the correctness of system behavior.

Embedded RTOSs are inherently concurrent and multitasking, where tasks synchronize and communicate via core kernel primitives including message queues, semaphores, events, and barriers. Correct implementation of these synchronization mechanisms is critical to guaranteeing deadlock-free execution, eliminating priority inversion, and satisfying stringent real-time constraints. Even subtle logical flaws or implementation defects can be activated under specific timing or interrupt-induced runtime scenarios, leading to unpredictable system behaviors[4]. Consequently, rigorous formal verification of RTOS code implementations is indispensable for establishing the functional correctness and operational reliability of multitasking applications deployed on top of the kernel.

RTOS code implementations are typically highly complex. Core kernel functions, including task management, synchronization, and inter-task communication, involve sophisticated concurrency control, interrupt processing, and resource scheduling, leading to strongly nondeterministic runtime behaviors. Consequently, conventional testing techniques fail to achieve

exhaustive verification, making them incapable of uncovering all latent code defects[5,6]. To overcome this limitation, formal methods have gradually become a primary approach for verifying operating system correctness[7,8]. Unlike traditional testing, formal methods construct operating system models using mathematical formalisms and provide rigorous definitions of correctness together with formal proofs, thereby theoretically establishing whether an operating system satisfies the required properties[6]. In recent years, formal methods have achieved remarkable progress in multiple critical research domains, including operating system kernel verification[9,10], safety-oriented design of the Rust programming language[11,12] and blockchain security verification[13,14]. These advances demonstrate the value of formal techniques in guaranteeing the functional correctness and operational safety of complex software systems.

However, the application of formal methods to verifying embedded RTOSs still faces numerous challenges. First, a full embedded RTOS has a large codebase and complex implementation details, and directly constructing a full model of the RTOS incurs substantial effort and high formal verification costs. Consequently, formal verification of embedded RTOSs has not yet become mainstream in industrial practice. Given practical cost constraints, only a subset of critical modules of an embedded RTOS is typically selected for formal verification[15]. This however introduces another challenge: when verifying critical modules, formal verification is usually performed independently, without fully considering the scalability and reusability of abstract models, which makes the verification of these critical modules a one-off effort[16]. Moreover, formal verification typically demands specialized expertise, raising barriers to its industrial adoption. In addition, efficiently establishing and verifying the refinement relation between constructed abstract models and their concrete implementations in a cost-effective manner remains a notable research challenge. References [17] and [18] offer insights into tackling this challenge, demonstrating that integrating formal analysis with automated tools can substantially reduce manual intervention.

In summary, although existing studies have achieved substantial advances in formal verification and modeling, the usability and scalability of formal verification techniques, together with the guarantee of model-implementation consistency, remain key bottlenecks that constrain their large-scale adoption during the practical development of industrial-grade embedded RTOSs. To address these challenges, this paper proposes a compositional modeling and verification approach based on the BIP component-based formal framework for our in-house embedded real-time operating system. Our main contributions are as follows:

(1) We propose a compositional modeling technique with scalability and usability, and we characterize the class of requirements it is able to discharge and the cost at which it does so (RQ1). The technique reduces modeling complexity and allows developers to assemble formal models incrementally from modules that have already been verified.

(2) Using refinement techniques, we establish observable equivalence between our abstract models and the corresponding code implementations over the generated traces, thus providing evidence of the correctness of system implementations; the equivalence is witnessed by generated kernel tests rather than asserted by inspection (RQ2). We also develop an automated test code generation tool that removes hand-written conformance tests from the loop.

(3) We perform full-process industrial-grade verification on our proprietary embedded real-time operating system and report what the workflow costs and what it returns in that process (RQ3). The results show that the method can be deployed within a real industrial product

environment, and they also delimit where it cannot yet be applied.

1.1 Research questions

The challenges discussed above can be condensed into a single engineering question: can formal verification of an embedded RTOS be organized so that it survives continuous modification of the kernel, instead of remaining a one-off exercise performed once per project? We decompose this question into three research questions, and the remainder of the paper is organized around them rather than around the individual techniques.

RQ1 (Tractability and fidelity). Can a kernel module of an industrial embedded RTOS be modeled at the requirement level in BIP such that property verification remains tractable, while the model still captures the scheduling, synchronization and timing behavior on which the target real-time properties depend?

RQ2 (Model-implementation consistency). Can behavior traces obtained from a verified model be converted automatically into kernel tests that establish observable equivalence between model and implementation, and which classes of model-implementation divergence does such a comparison expose?

RQ3 (Engineering value). What does the workflow cost and what does it return in an industrial development process, in terms of the modeling artifacts that must be maintained, the effort needed to regenerate tests after a kernel change, and the reusability of verified modules across scenarios?

Section 4 shows how the method answers each question by construction: compositional atomic modules and requirement-level properties address RQ1, the tagged trace protocol and the refinement dictionary address RQ2, and the separation of model, observer and dictionary artifacts addresses RQ3. Section 5 answers the questions empirically and states the limits of each answer. We do not claim exhaustive verification of a whole kernel; Section 5.5 makes the scope of the claims explicit.

The remainder of this paper is structured as follows. Section 2 reviews the related work. Section 3 introduces the background. Section 4 presents the proposed method and shows how each research question is addressed by design. Section 5 reports the experimental evaluation, answers RQ1 to RQ3 in turn and discusses the threats to validity. Section 6 concludes and outlines future work.

2 Related work

seL4[9], the world’s first high-performance microkernel with source-code-level formal verification, establishes a rigorous refinement chain from high-level abstract specifications to C and assembly implementations, covering core OS functionalities including system calls, interrupt handling, and thread management. It represents a milestone in OS formal verification research. Building upon this foundation, researchers have extensively explored operating system formal verification and achieved substantial progress in recent years.

Gu et al.[19] from Yale University developed CertiKOS, a formally certified kernel tailored for secure cloud computing, to mitigate information leakage security risks inherent in traditional cloud virtual machine server systems. By decoupling resource management from resource delegation, the team constructed a lightweight, modular trusted kernel. Formal verification tools were integrated throughout the kernel design process to validate its functional correctness and information flow security. On the basis of their preliminary research, Gu et al.[20] further proposed

a context-refinement-based hierarchical verification framework for concurrent system scenarios. By systematically modeling hierarchical abstraction layers, environmental contexts, and atomic system objects, they implemented the first fully verified concurrent operating system kernel and completed its end-to-end functional correctness verification using the Coq proof assistant.

Xu et al. [21] presented a practical verification framework for preemptive operating system kernels built upon interrupt verification theory and concurrent program refinement theory. This framework was applied to the commercial real-time kernel μC/OS-II to verify the functional correctness of critical modules, including the scheduler, timer interrupt handling, message queues, semaphores, and mutex mechanisms.

Targeting the ARINC 653 separation kernel standard, Zhao et al.[22] established formal functional specifications and an information flow security model in Isabelle/HOL via refinement-based methods. They also uncovered several security vulnerabilities in practical kernel implementations through code auditing. Following this work, Zhang et al.[23] adopted Event-B to construct a seven-layer refined specification for multi-core ARINC 653 systems. Their work verified the internal consistency of the standard specification and extended formal analysis capabilities to multi-core partitioned operating systems.

Sun et al.[24] built a multi-layer verification framework for Global Platform-compliant embedded operating systems. This framework covers hierarchical formal verification layers ranging from formal requirement definitions, high-level and low-level specifications, down to the assembly implementation layer. Chong and Jacobs[16] adopted VeriFast to verify the memory safety, thread safety, and functional correctness of concurrent queues — the core inter-process communication mechanism of FreeRTOS — and detected multiple latent functional defects. For the RTEMS operating system, Shi et al.[4] leveraged Frama-C to formally verify the implementations of the Immediate Ceiling Priority Protocol (ICPP) and the Multiprocessor Resource Sharing Protocol (MrsP). Their work identified several inconsistencies between theoretical designs and practical implementations and further proposed targeted remediation solutions. Butterfield and Tuong[25] employed the Promela/SPIN model checker to generate test cases for RTEMS kernel synchronization modules. They realized a mapping from abstract model behaviors to concrete C test codes by adopting a dedicated observation language.

Evrard and Donaldson[26] adopted Promela and SPIN to verify multiple futex mutex and condition variable implementations corresponding to Linux futex system calls. Their work successfully reproduced known historical defects and validated the correctness of mainstream implementations, while evaluating the practical effectiveness of symmetry reduction and memory compression optimization techniques. Carbonneaux et al.[27] leveraged the Iris verification framework to formally verify two inter-process communication queue data structures for operating systems. Through rigorous formal verification, they improved the reliability of iteratively updated industrial-grade code and evaluated the practical cost-effectiveness of formal methods in industrial scenarios. Their verification work formally proves the functional correctness of queue algorithms, identifies and fixes bugs in multiple device drivers, simplifies the original algorithm logic, and optimizes overall system performance. Furthermore, their practical findings have promoted the industrial adoption of formal verification techniques within Meta.

Taken together, these studies expose three gaps when measured against the research questions of this paper. With respect to RQ1, the strongest results come at the price of whole-kernel proofs (seL4, CertiKOS) whose cost is prohibitive for an in-house kernel under continuous modification,

while the model-checking studies that remain tractable target isolated mechanisms rather than a requirement-level, layered component model that keeps the state space bounded. With respect to RQ2, most model-checking work stops at the model; the few trace-generation approaches (Butterfield and Tuong) do not fully address observable equivalence or the localization of a divergence to a single recorded action. With respect to RQ3, very little of this literature reports the maintenance, reuse and change cost of the artifacts under iterative kernel development, which is what decides whether verification survives a release cycle. These gaps are compounded by the fact that conventional OS formal verification workflows are extremely labor-intensive and incur excessive verification overhead[9,10,15], which is unaffordable for the industrial development of our in-house RTOS product. To address these limitations and facilitate the practical deployment of formal verification in industrial-grade embedded systems, this paper presents a complete formal verification workflow for our in-house RTOS based on the BIP modeling framework.

3 Background

The approach proposed in this paper targets our in-house embedded real-time operating system and leverages the BIP formal modeling framework. Relevant concepts and fundamental background knowledge are introduced in the following section.

3.1 In-House Embedded Real-Time Operating System

The embedded real-time operating system (RTOS) considered in this work is an in-house system, hereafter referred to as the RTOS. It adopts a microkernel architecture with tailorable and configurable modules, and its minimum system code size is less than 20 KB. The RTOS employs event-driven, priority-preemptive scheduling, achieving microsecond-level interrupt response, task switching, and task response times that are comparable to those of mainstream commercial products. It provides native APIs, and POSIX interfaces to facilitate the rapid and seamless migration of existing applications. The system supports general-purpose processors including Phytium and Loongson, DSPs such as Phytium and Tianhua, and Tianrui series SoCs. Currently, the RTOS has been deployed in multiple application domains and maintains stable, reliable operation.

The task management module serves as the core component for scheduling and resource management in this in-house RTOS. It implements full-lifecycle task management, covering task creation, activation, scheduling, execution, dynamic priority adjustment, suspension and resumption, synchronous coordination, and deletion. This comprehensive management mechanism guarantees rigorous real-time performance and behavioral predictability in concurrent multitasking scenarios. The system adopts fixed-priority preemptive scheduling as its core task scheduling policy, where a smaller numerical priority value indicates a higher task priority. Each task is equipped with an independent Task Control Block (TCB) that records core task information, including task ID, current priority, running state, remaining execution time, and occupied synchronization resources. For multitask synchronization, the module provides semaphore and mutex primitives. In particular, the mutex mechanism integrates the Priority Inheritance Protocol and Priority Ceiling Protocol to effectively resolve the priority inversion issue. As a critical submodule of task management, clock management relies on periodic clock ticks to handle task timeout waiting, cumulative execution time statistics, and periodic task scheduling. This capability ensures that the execution duration of each task remains well constrained and effectively avoids task deadline misses.

3.2 The BIP Component Framework

BIP (Behavior-Interaction-Priority) is a component-based formal framework developed by Verimag, France, for the rigorous design and modeling of complex reactive systems. It integrates a dedicated formal modeling language and a complete supporting toolset to enable correctness-by-construction system development. This paradigm avoids the expensive, error-prone verification efforts that conventional workflows require late in development. The design philosophy of BIP is to construct systems as coordinated compositions of atomic components, emphasizing the separation and rigorous definition of three layers: behavior, interaction, and priority. The behavior layer defines the internal behavioral logic of system components; the interaction layer defines synchronization protocols among components via connectors to describe their synchronization and communication mechanisms; the priority layer imposes dynamic filtering rules over interactions to express scheduling policies, conflict resolution, and performance constraints.

A BIP model is typically organized at three levels: atomic components, connectors, and compound components. Atomic components serve as the basic units, containing behavior descriptions, port interfaces, and data variables. Connectors define interaction patterns and priority rules to realize synchronization and coordination between components. Compound components encapsulate multiple subcomponents into higher-level modules through hierarchical composition, supporting the recursive construction of complex systems.

The execution of a complete BIP program is driven by a port-triggering mechanism and scheduled at runtime by the BIP Execution Engine. In addition, BIP offers a rich toolchain and supports seamless interoperability with C/C++, enabling models to undergo formal analysis and be directly compiled into efficient executable code. SBIP [29] extends BIP by incorporating statistical model checking mechanisms, which enables statistical model checking of safety properties for formal models built within BIP.

In recent years, the BIP framework has been extensively adopted in the development and verification of complex safety-critical systems, demonstrating its practicality and engineering feasibility. In the domain of network protocol security, BIPSec[30] and IKEv2-FV[31] leverage BIP to conduct end-to-end verification of AH/ESP modules in industrial IPsec protocol stacks and state machine modeling for the IKEv2 key exchange protocol, respectively. These studies demonstrate BIP's powerful capability in handling protocol concurrency, priority constraints, and semantic consistency between formal models and concrete implementations. In the field of aerospace embedded systems, BIP has been integrated with the NASA F Prime framework[32]. Through component transformation and state-space verification, BIP can capture system-level interaction errors and subtle race conditions that are inaccessible to conventional testing methods, and its real-time deployment capability has been validated on Raspberry Pi platforms. For concurrent software verification, JavaBIP combined with VerCors extends the BIP coordination paradigm to support deductive verification of data-related properties in Java concurrent programs[33]. In robot behavior modeling, a compositional transformation scheme from behavior trees to BIP empowers traditional behavior trees with formal verification functions, enabling automatic detection of potential design defects[34]. These cross-domain application cases demonstrate BIP's value in improving the reliability and verifiability of complex industrial systems.

Furthermore, the modular nature of BIP closely matches the structure of the task management module of our in-house embedded real-time operating system. This enables core kernel operations,

including task creation, activation, scheduling, suspension and resumption, priority adjustment, and synchronization and mutual exclusion, to be naturally mapped into independent atomic behavioral components. Meanwhile, synchronous interactions among components are explicitly specified via ports and connectors. Therefore, we adopt the BIP framework for formal modeling and verification of our in-house embedded real-time operating system.

4 BIP-based formal verification of an embedded real-time operating system

4.1 Overall Framework

Figure 1 illustrates the proposed formal verification framework for embedded real-time operating systems. The framework comprises two primary phases: formal modeling and model refinement verification. In the formal modeling phase, the BIP component framework is used to construct formal models for several critical modules of the in-house embedded RTOS. Model checking tools verify whether the formal model satisfies predefined invariants. In the model refinement verification phase, execution traces generated from the formal model are converted into executable test cases. This step validates the refinement relation between the formal model and the concrete implementation of the in-house RTOS, ensuring consistency between model specifications and actual kernel execution.

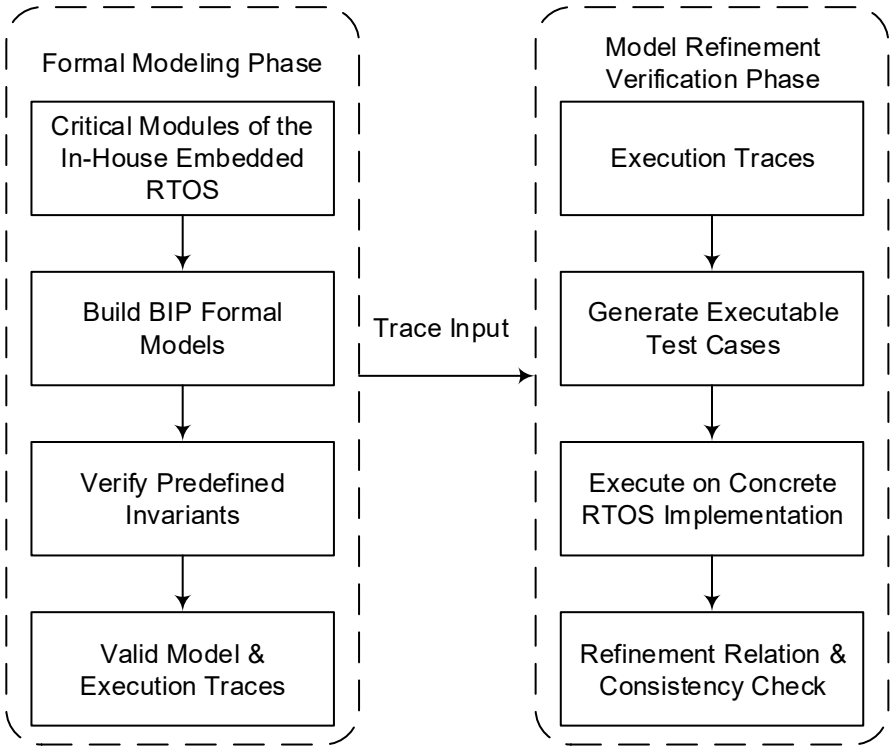

Fig. 1. Architecture of the proposed approach

4.1.1 Formal Modeling Approach

Given that our in-house RTOS is continuously updated with iterative code modifications, it is crucial to apply formal verification techniques to such evolving software systems[35]. To enhance the generalizability of formal verification workflows, this paper performs requirement-level modeling and verification for the embedded real-time operating system. We assume that the requirements specification of an embedded real-time operating system is modified far less frequently than its algorithmic implementations and source code. Therefore, this work focuses on

formal modeling based on the requirements specification of the in-house embedded RTOS.

Following modular and compositional verification principles and considering the architectural and functional features of the in-house RTOS, we decompose the entire system into fine-grained functional modules, covering task management, semaphore mechanisms, and message queue modules. Each module is verified independently at the requirement level, so that the state space that has to be analyzed stays bounded per module and per horizon; this is the concrete way in which the decomposition serves RQ1. The composition rules of Definitions 3 and 4 then provide a design framework that organizes already-verified modules into layers, rather than a claim that system-level verification has been completed: system-level composition verification is left to future work. The overall verification methodology and technical logic are elaborated in the following sections.

In the single-module verification workflow, the BIP abstract behavioral modeling language is used to formally characterize the system behaviors and interface specifications on which module M depends, constructing an abstract behavioral model Ψ. Linear temporal logic is utilized to formalize the verification objectives of the module, yielding a set of formal target properties Φ that can be validated against the model Ψ. Subsequently, we adopt an abstract behavior simulation engine, a statistical model checking engine, and a behavior analysis engine to jointly verify the consistency between the abstract behavioral model Ψ and the target functional properties Φ. Finally, guided by refinement techniques and module interface specifications, we establish the mapping between the abstract module behavior and its concrete execution behavior and verify their consistency. This procedure ensures the traceability and consistency from high-level target properties down to the concrete system implementation. Since the verification only targets the abstract behavior of a single independent module, the state space and execution paths generated during verification remain manageable. This effectively mitigates key limitations of automated verification, including state explosion and path explosion.

The definition and verification method of atomic modules are given below.

Definition 1 (Atomic Module): An atomic module is a software module with relatively independent functionality, formally defined as $\mathrm{M} = \Psi_1 \rightsquigarrow_R \mathrm{C}: \Psi_2$, where:

(1) Ψ_1 is the specification of the underlying API on which atomic module M depends, and Ψ_2 denotes the API that M provides to the outside.

(2) Both Ψ_1 and Ψ_2 represent a state transition system (S,T), where S denotes the set of program states, including the global system state and the input/output states of the system API; T denotes the set of state transition relations, of type $P(S \times S)$, where P denotes the power set.

(3) C is the code implementation of atomic module M.

(4) R represents the refinement relation between C and Ψ_2.

The relevant logic for atomic module verification and the composition verification of multiple atomic modules is defined below.

Definition 2 (Verified Atomic Module): Given an atomic module M, a verified atomic module is defined as $M \rightsquigarrow_R C: \Phi$, where C is the code implementation of atomic module M, and R represents the refinement relation between C and Φ.

A verified atomic module can be interpreted as meaning that the code implementation C of M strictly satisfies its target properties Φ.

When modular verification is employed, the atomic modules of the system must be

partitioned into different abstraction levels. Only atomic modules with the same abstract state and refinement relation can be composed at the same abstraction level. This partitioning offers the following two advantages:

(1) Atomic modules in each layer can be reasoned about and verified individually without concern for inter-module interaction dependencies. Moreover, within a specific domain, verified atomic modules can be composed into a larger verified atomic module, forming a verified hierarchical state machine.

(2) Hierarchical partitioning can transform complex interaction dependencies among arbitrary modules into concise vertical call relations among hierarchical state machines, thereby significantly reducing the workload of reasoning and verification for the entire system. Figure2 provides an intuitive illustration of this explanation.

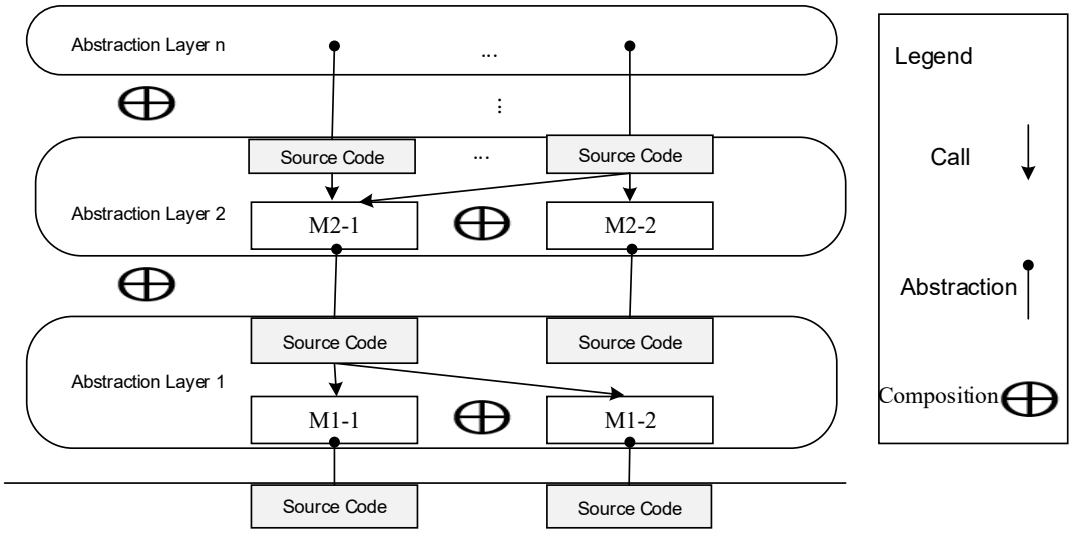

Fig. 2. Compositional verification schematic

The formal description of the composition verification rules is given below.

Definition 3 (Parallel Composition): Given two atomic modules $M_1 = \Psi \nrightarrow_R C_1 : \Psi_1$ and $M_2 = \Psi \nrightarrow_R C_2 : \Psi_2$, if M_1 and M_2 have the same abstract state and refinement relation, then M_1 and M_2 can be composed in parallel to form a verified larger atomic module$M$, where $M = M_1 \oplus M_2, \Psi \nrightarrow_R C_1 \oplus C_2 : \Psi_1 \oplus \Psi_2$.

Definition 4 (Vertical Composition):Given two atomic modules $M_1 = \Psi_1 \nrightarrow_R C_1 : \Psi_2$ and $M_2 = \Psi_2 \nrightarrow_R C_2 : \Psi_3$, if M_2 needs to invoke the interface model of the lower-layer state machine M_1, then M_1 and M_2 can be vertically composed to form a larger atomic module M, where $M = M_1 \oplus M_2, \Psi_1 \nrightarrow_R C_1 \oplus C_2 : \Psi_2$.

From Definitions 3 and 4, it can be seen that the parallel composition rule allows atomic modules at the same abstraction layer to be composed together into a verification layer, while the vertical composition rule allows verified modules at different abstraction layers to be composed into a layered verification structure. These rules define what a sound composition looks like; the resulting system-level composition verification is not performed in this paper and is left to future work.

The composition rules come with explicit preconditions and carry a corresponding risk. Parallel composition requires the two modules to share the same abstract state and the same refinement relation, and vertical composition requires the upper module to invoke the lower module only through its interface state machine; a composition that violates either condition does

not preserve the properties of its parts. The property-preservation argument is therefore an assume-guarantee one: each module guarantees its target properties under the assumption that its neighbors respect the recorded interface contracts. This is the sense in which Definitions 3 and 4 are a design framework: they state when composition is sound, not that every composition has been checked, and the system-level composition verification that would close the argument is left to future work.

4.1.2 Model Refinement Approach

After the formal model is established, we first use the BIP statistical model checking engine to verify the consistency between the target properties and the abstract model. We then obtain each behavior trace that passes property verification (such traces are examples of correct system operation). Based on the principles of refinement techniques, we automatically generate kernel test code from these behavior traces and run the tests on a multi-core simulator or a physical machine. If the system passes the tests, this is evidence of observable equivalence between the abstract model and the concrete implementation over the generated traces, not a proof that they cannot be distinguished in every possible context. Here, observable equivalence means that the model and the implementation agree on every observable record of every generated scenario. The following further explains the consistency verification between the abstract model and the concrete implementation based on the principles of refinement techniques in the field of formal verification (as shown in Figure 3 below).

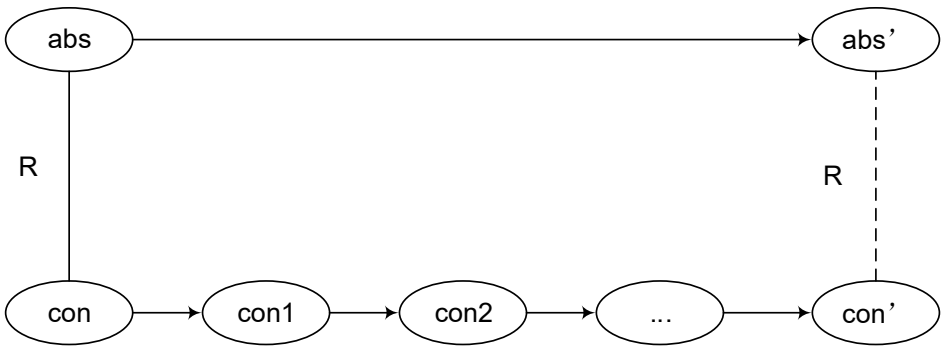

Fig. 3. Illustration of the refinement principle

In Figure 3, $abs \rightarrow abs'$ represents a one-step state transition in the abstract model. It is a high-level description of the system that focuses on the overall functionality and behavior while ignoring concrete implementation details. In contrast, $con \rightarrow con'$ usually represents a multi-step state transition in the concrete implementation. It is the actual implementation of the system and contains more details and intermediate steps. The relation R in Figure 3 is a binary relation defined between two systems or states to represent a data refinement relation. If abs and con satisfy R, denoted by $absRcon$, this means that the behaviors of abs and con are equivalent with respect to R.

The consistency guarantee between the abstract system and the concrete implementation is explained as follows: if $\text{abs} \rightarrow \text{abs}'$ is a one-step abstract behavior and abs and con satisfy R, then the concrete behavior must satisfy the following condition: starting from con, after several transition steps, it can eventually reach a state con' such that con' and abs' also satisfy R.

Two specification mechanisms are used, because the target properties of Table 1 do not all have the same shape. Properties that constrain a state predicate over time are specified as an

observer function together with an LTL formula. The observer is written in the C++ support layer: one_running() counts the tasks that are in the running state, task_timeout() compares the accumulated cost time of every task with its period, and the mutex and priority observers inspect the holder and the effective priority of each binary semaphore. An observer is exported from the model by trace_b("Scenario.p", value), which emits the record var bool Scenario.p true|false at every step in which the observer is evaluated. The LTL formula then constrains that exported variable, for example G{1000} (Scenario.taskTimeOut). The bound n in G{n} is the horizon of the property, expressed in steps of one sampled run: unbounded liveness cannot be decided by statistical model checking, so each property is stated over a horizon that exceeds the longest scenario of the module, which is 1,000 steps for the task and semaphore models and 500 for the heap model. Properties that concern progress and termination are discharged structurally instead, by the guards of the test-driver state machines (Section 4.2.3): a driver can leave its waiting state only when isTasksFinished() holds, so a run that never terminates is exactly a run in which the module under test fails to make progress. Section 5.2 reports which mechanism discharges which property, and why the distinction matters for the strength of the resulting evidence.

4.2 Formal Modeling

In this work, we formally model several critical modules of the in-house embedded real-time operating system (RTOS), including task management, memory management, message queues, semaphores, events, and barriers. Due to space limitations, we focus on task management as an illustrative example.

A task is the basic unit of scheduling in the in-house RTOS. It is referred to as a thread at the kernel level and as a task at the user interface level (also called a thread in the POSIX API); these terms denote the same entity. Task management in the in-house RTOS follows a layered design: the kernel-level task management module implements the core task management functions, while the user interface layer encapsulates them and exposes task management APIs to users.

The multi-core support of the in-house RTOS significantly increases code complexity, especially in task scheduling. Although the in-house RTOS has detailed design documents and has undergone extensive testing, it has not yet been subjected to rigorous formal verification of critical properties or to broader-scenario code testing based on verified behavior traces. To address this, we adopt the user interface layer APIs as the basic functional verification units. By finely modeling the task management APIs, system scheduling, and clock control, we formally verify the functional correctness and safety properties of the kernel thread management module.

Based on the core design requirements of the in-house RTOS, we identify a set of properties, listed in Table 1, that must be verified to ensure the real-time determinism and reliability of the task management module. Verifying these properties provides effective evidence that the task management module maintains correct behavior under various boundary conditions, thereby providing a theoretical foundation for the real-time determinism and reliability of the in-house RTOS.

Table 1. Target properties of the in-house embedded RTOS

No.	Property Type	Detailed Requirements
1.	Target Safety Properties	Deadlock-free task scheduling: The system never enters a deadlock state.
2.		Task scheduling safety: At any time, only one task is in the running state.

3.		Task execution timeliness: The total execution time of a task (including CPU execution time, resource waiting time, and scheduling time) must not exceed the task period.
4.		Task priority protection: If a task currently holds any binary semaphore governed by the locking protocol, its priority must not be lowered immediately.
5.		Data access consistency: At any time, a mutex semaphore can be held by at most one task.
1.	Target Functional Properties	Verification of invalid task creation
2.		Verification of invalid task deletion
3.		Verification of valid task creation and deletion
4.		Verification of invalid task activation
5.		Verification of valid task activation
6.		Verification of invalid task suspension
7.		Verification of valid task suspension
8.		Verification of invalid task resumption
9.		Verification of valid task resumption
10.		Verification of priority lowering for a task holding a mutex

As shown in Figure 4, the BIP model developed for the task management module of our in-house embedded real-time operating system (RTOS) comprises two main parts: an upper-layer task abstraction model and a lower-layer supporting model. The upper-layer task abstraction model mainly consists of an API model, a top-level compound model, a scenario control model, four test driver models, two worker task models, a system model, and a clock model. The lower-layer supporting model performs task execution, priority preemption, semaphore and mutex synchronization, random scenario generation, and property checking, thereby verifying the correctness of the modeled properties.

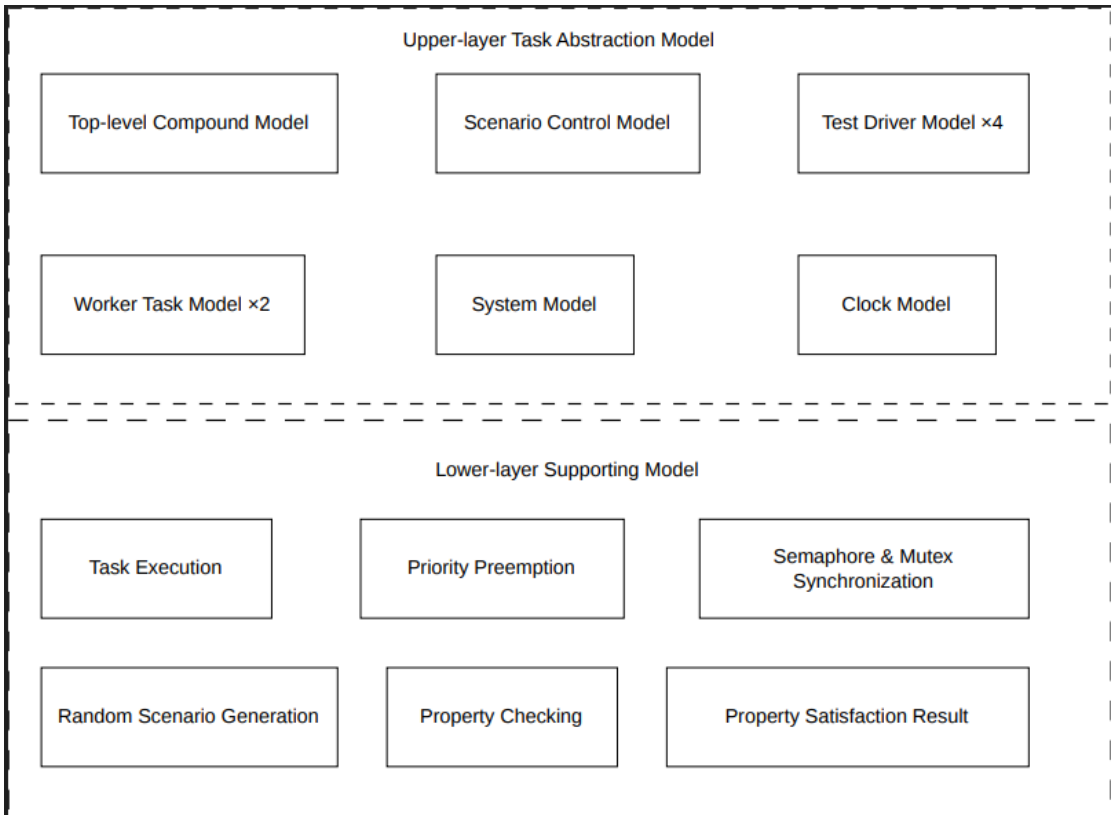

Fig. 4. Task management module model of the in-house embedded RTOS

4.2.1 Top-Level Compound Model

The top-level compound model integrates a scenario control model, four test driver models, two worker task models, a system model, and a clock model. Using dedicated connector types, it establishes port bindings among these models and integrates them into a fully executable system. In addition, multiple property-tracking variables are introduced into the scenario control model to record critical correctness states of the system in real time. Within the top-level compound model, the submodels interact closely through well-defined ports and connectors to jointly execute tests for different scenarios and simulate the behavior of the task under test. Owing to the features of the BIP framework, both internal and external ports of the model can be scheduled randomly, which facilitates modeling interruptible systems. The model interactions support modeling both synchronous and asynchronous event-triggered relations.

4.2.2 Scenario control model

The scenario control model serves as the central control component of the overall architecture. It is responsible for global initialization, scenario selection, and property monitoring. It maintains shared data such as task IDs, system identifiers, clock identifiers, and semaphores, and, during initialization, outputs definition and declaration information, generates random scenarios, and performs initial property checking. Figure 5 presents the design of the scenario control model.

The scenario control model achieves dynamic coordination and property verification of the entire system through a state machine and port interaction mechanisms. Upon model startup, it first enters the initial state and assigns global variables, including initializing task identifiers, system identifiers, clock identifiers, and mutex identifiers. It also calls external functions to output

definition and declaration information, including the corresponding information formats. Based on the output of a random scenario generation function, it selects the verification path for the current run from four scenarios: creation and deletion, task activation, suspension and resumption, and priority change. Once the scenario is determined, the scenario control model triggers the corresponding test driver model to start through an export port. During the execution of the test driver model, the scenario control model continuously monitors internal transition ports and task ports, and synchronously waits, via semaphores, for completion signals from the test driver model and worker task models, thereby ensuring smooth execution of the overall system flow.

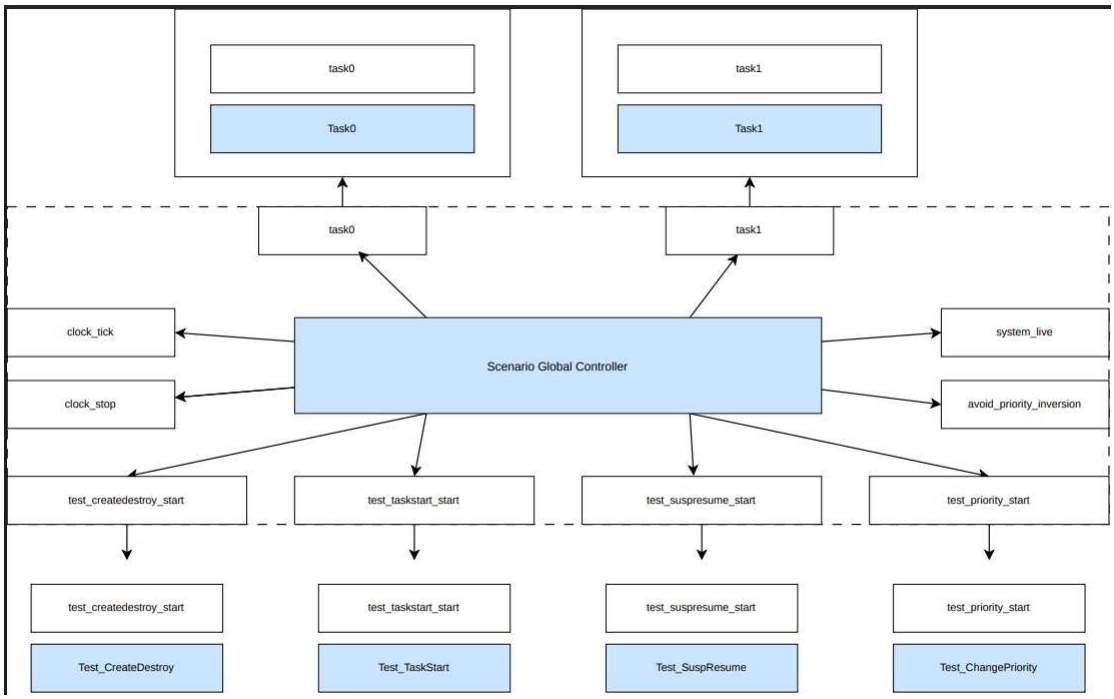

Fig. 5. Scenario control model

4.2.3 Test Driver Model

The four test driver models are designed for the creation and deletion, task activation, suspension and resumption, and priority change scenarios, respectively. Each test driver model employs an internal state machine to sequentially perform parameter setup, task creation, activation, synchronization waiting, and cleanup, and communicates with the worker task models via semaphores to coordinate the execution order.

The test driver model for the creation and deletion scenario primarily handles task allocation and reclamation. After receiving a start signal from the scenario control model in its initial state, it first binds the task parameters and then decides whether to perform a creation or deletion operation based on the current scenario flag. The creation operation invokes the underlying creation function to instantiate worker task models and remains in a waiting state until the tasks complete. Once it detects that all tasks have finished, it performs the deletion operation and finally releases the task resources it holds. Through a self-loop waiting mechanism, the model ensures that cleanup is performed only after all tasks have truly completed their work.

The test driver model for the task activation scenario, after binding the parameters, sequentially performs task creation and activation. Once activation is complete, it waits via a semaphore for completion signals from the worker task models. Upon receiving the completion signal, it proceeds to the subsequent deletion and resource release phases. If the scenario requires

skipping task activation, it bypasses the activation step and proceeds directly to the deletion path. This design enables the model to cover both normal activation and abnormal non-activation cases, thereby thoroughly verifying the behavioral correctness of the activation interface in different contexts.

The test driver model for the suspension and resumption scenario first performs task creation and activation after receiving the start signal. It then either allows the worker task model to suspend itself or initiates suspension externally from the test driver, depending on the scenario configuration. After suspension, it repeatedly checks the state of the worker task model and, once successful suspension is confirmed, performs the resumption operation. Upon completion of resumption, it waits via a semaphore for the completion signal from the worker task model, inserts additional suspension and resumption steps as needed, and finally enters the cleanup phase to release all resources. This model implements multiple variants, including self-suspension, external suspension, and immediate suspension after activation, enabling comprehensive testing of operations related to suspension and resumption and ensuring that suspension and resumption are always correctly performed throughout the task lifecycle.

In addition to creating and activating a primary worker task model, the test driver model for the priority change scenario also creates and activates an auxiliary worker task model to establish a priority contention environment. During system execution, the model first completes the creation and activation of the primary task, then creates and activates the auxiliary task according to the scenario flag. It then handles possible self-suspension and resumption operations, waits sequentially for completion signals from the two worker task models via multiple semaphores, and finally, after all tasks have completed, performs priority checking and modification and completes the final deletion and resource release. Through nested task creation and activation flows and synchronization via multiple semaphores, the model precisely controls the concurrent execution timing of the two worker task models, thereby constructing verification scenarios for key mechanisms such as priority inheritance, priority promotion, and priority demotion.

4.2.4 Worker Task Model Design

The two worker task models, Task0 and Task1, simulate the behaviors of real RTOS tasks, including waiting for a start semaphore, task self-suspension, priority inspection, mutex acquisition and release, and the final release of the completion semaphore.

Task0 acts as the primary worker task model. After receiving the start signal, it enters a state to wait on the start semaphore. Once the semaphore is available, the model acquires and then releases it. Next, it determines whether to trigger self-suspension according to the scenario configuration. Depending on whether priority testing is needed, it further decides whether to enter the mutex acquisition routine. While holding the mutex, Task0 executes operations including priority inspection, priority reduction, and priority restoration. Finally, it releases the completion semaphore and transitions to the terminal state.

Task1 is the auxiliary worker task model, designed to cooperate with Task0 to build the priority contention environment. It also begins by waiting for the start semaphore, followed by semaphore acquisition, mutex acquisition, priority elevation check, mutex release, and completion semaphore release. Its core function is to compete with Task0 for mutex ownership in the priority change scenario, which verifies whether priority elevation applied to a low-priority task by a high-priority task functions correctly.

4.2.5 System and Clock Model Design

The system model and clock model provide essential environmental support for the entire BIP task management model. The system model periodically wakes tasks that reside in the waiting state, while the clock model processes timeout events and records task execution time via system clock ticks. Cooperatively, the two models simulate the background scheduling logic and time advancement mechanisms of the in-house RTOS, enabling accurate temporal and environmental modeling for formal verification of task scheduling behaviors.

4.2.6 Lower-layer supporting model

To accurately capture the realistic task behaviors of the in-house RTOS, dedicated mechanisms are required to simulate kernel scheduling operations and other state-update behaviors that govern task execution. For this purpose, we design the lower-layer supporting model illustrated in Figure 5. Leveraging the external C++ invocation interface of the BIP framework, this model constructs an OS service simulation layer based on core kernel data structures, including task control block arrays, priority ready queues, binary semaphore tables, and global clock variables. It integrates functional modules covering state machine management, priority scheduling, synchronization and mutual exclusion, clock timing, and random scenario generation. Furthermore, it provides standardized simulated system-call APIs to support the operational execution of upper-layer BIP models.

The lower-layer supporting model serves as a critical adaptation layer that bridges abstract OS service logic and concrete BIP model implementations. On the one hand, it encapsulates native RTOS system calls—including task creation, deletion, activation, suspension, resumption, and priority adjustment—into lightweight simulation functions that can be directly invoked by BIP atomic components. On the other hand, it exposes internal state query interfaces, such as task ID acquisition, queue emptiness detection, and current running task retrieval, enabling the top-level BIP coordination model to monitor and steer the entire simulation process in real time. Benefiting from these designs, the proposed BIP model supports precise statistical model checking for the task management module without deploying physical hardware or running the complete in-house RTOS kernel.

4.3 Model Refinement

Model building and refinement play two different roles and produce two different kinds of evidence. Behavior simulation and statistical model checking executed by the BIP engine validate the model against the target properties of Section 4.2, which confirms that the requirements of the in-house RTOS are satisfied at the requirement level; this evidence answers RQ1 and is reported in Section 5.2. Separately, the engine is used as a scenario enumerator: every run that reaches a terminal state is an execution trace that the kernel has been shown to admit, and those traces are the input of the refinement step. The traces are converted into test code which is executed against the concrete kernel, so that agreement between the two is evidence of observable equivalence rather than of syntactic similarity. This second kind of evidence answers RQ2 and is reported in Section 5.3.

To realize automatic transformation from abstract models to functionally verified code with observable equivalence, we adopt the refinement method presented in[20] and build a YAML mapping dictionary that defines the refinement rules linking abstract models to concrete implementation code. We also develop an automatic transformation tool named bip2c. This tool

converts execution trace exploration outputs generated by the model checker for the formal model into compilable, runnable concrete test code, which validates the consistency between the abstract model and its concrete implementation.

Within the BIP model, test actions and their corresponding parameters are explicitly defined by embedding tagged directives following a custom format (lines prefixed with "@@@"). Each tagged directive is a record of the form @@@ <bid> <TYPE> <payload>. The field bid identifies the BIP atomic component that emitted the record, which is what allows a generated test to reproduce the interleaving of components that produced the trace: two components that invoke the same API are distinguished by bid even when their arguments coincide. The field TYPE classifies the record, and the payload carries the action name and the actual arguments for a call, the name and the current value of a variable, or a task state. Twelve record types are used, falling into four groups: declarations (INIT, DEF, DECL, DCLARRAY, NAME) that fix the constants and the objects of the scenario; task context (TASK) that names the task executing the current segment; API invocation (CALL); and observable effects, namely SCALAR for the value of a variable after a call, typically the returned status code, STATE for a task-state transition, SIGNAL and WAIT for synchronization operations, and LOG for free annotations. A call and its outcome are therefore two consecutive records rather than one composite tuple, as in @@@ 4 CALL task_create 1 5 0 0 2 createRC followed by @@@ 4 SCALAR createRC 0. The record stream serves both as the runtime log of the model and as the atomic unit parsed by the generator. This contract-based modeling approach embeds test intent directly within the formal model. Consequently, any counterexample trace or reachable path produced by the model checker during state-space exploration can be converted into complete test steps by extracting the tagged sequence. This strategy eliminates the semantic gap between models and test scripts that exists in conventional approaches[20].

Keeping the invocation and its effect in separate records is what makes the later comparison granular: the CALL record fixes the order and the arguments, the SCALAR record fixes the effect, and either can disagree on its own. The same separation is what allows a trace to be replayed without ambiguity, because the arguments that a generated test has to supply are exactly the payload of the CALL records, and the values it has to observe are exactly the payload of the SCALAR and STATE records.

To translate the abstract primitives of the BIP model into executable target code, this work adopts a declarative refinement mapping dictionary stored in YAML format. The dictionary defines one code template for every action type that can occur in a CALL record, covering task creation, activation, suspension and resumption, priority change, and semaphore and mutex operations. Each template reserves positional placeholders, marked as {0}, {1} and so on, for the arguments of that action; the template for task creation reads as follows.

```
task_create: |
    LOG("Calling TaskCreate(%d,%d,%d,%d,%d)", {0}, {1}, {2}, {3}, {4});
    rc = task_create({0}, {1}, {2}, {3}, {4});
    LOG("Returned 0x%x from Create", rc);
```

During generation the placeholders are replaced by the actual arguments of the corresponding CALL record, so that the concrete invocation is derived from the verified one instead of being written by hand. The surrounding test language is configuration as well: a language descriptor supplies the segment naming scheme, the segment declaration and body delimiters, the logging

and assertion macros, and the comment syntax. Porting the generator to another target language or another test harness therefore requires a new descriptor and a new dictionary, not a change to the generator itself.

The generated artifacts are copied into the test directory of the in-house RTOS and integrated into its build pipeline by updating the corresponding configuration file, after which the test executable is rebuilt and executed. Each trace becomes one test segment: a function that receives the test context, replays the recorded CALL records in order on the concrete kernel, and checks every SCALAR, STATE, SIGNAL and WAIT record with the assertion macros of the harness. Two properties of this construction matter for the interpretation of the results. First, a divergence does not have to be searched for afterwards: the failing assertion identifies the segment, the position of the record and the expected value, so the mismatch is localized to one invocation in one scenario. Second, because the replay is driven by bid and by the recorded argument values, the test is deterministic with respect to the modeled interleaving, which is what makes a trace an executable witness rather than an illustration. Any inconsistency therefore surfaces as a test failure at the corresponding location.

4.4 From specification to evidence

Sections 4.1 to 4.3 describe a method, but they also make three claims that can be falsified. First, a requirement-level module model is small enough for statistical model checking to terminate within an ordinary engineering budget, while still exposing the scheduling and synchronization behavior over which the properties are quantified. Second, the record stream produced by such a model is rich enough to drive the concrete kernel, so that agreement between the two is evidence of observable equivalence and not merely of syntactic similarity. Third, the artifacts that carry the verification argument, namely the model, the observer functions and the refinement dictionary, are separable, so a verified module can be reused and regenerated without repeating the argument. Section 5 tests these three claims in order and reports, for each, not only whether it holds but what its holding implies for a kernel engineer.

5 Experimental evaluation

To evaluate the effectiveness of the proposed approach, all experiments are conducted on a VMware virtual machine. The detailed experimental configuration is listed in Table 2.

Table 2 Experimental environment configuration

No.	Configuration item	Software name
1	Virtual machine software	VMware Workstation 17 Pro
2	Virtual machine operating system	Ubuntu 24.04 LTS
3	Host operating system	Windows 10

The evaluation is organized around the research questions rather than around the individual techniques. Section 5.1 fixes the experimental setup and defines the metrics used throughout; Section 5.2 answers RQ1 with the property-level results of the task-management module; Section 5.3 answers RQ2 with the refinement and consistency results; Section 5.4 answers RQ3 with the scale of the artifacts and the cost of regenerating them; Section 5.5 discusses the threats to the

validity of these answers. The task-management module is used as the running example because it exercises scheduling, synchronization and timing simultaneously, while the remaining modules are reported at the level of their property inventory.

5.1 Experimental setup and evaluation metrics

The target properties are formalized in linear temporal logic (LTL) over the variables exported by the observer functions described in Section 4.1.2, and are checked by statistical model checking (SMC) of the BIP model. The engine is configured with an error probability $\alpha = 0.001$ and a precision $\delta = 0.05$: the estimated probability of a property deviates from its true probability by at most δ with confidence $1 - \alpha$, that is, with probability of at least 0.999. Each configuration is sampled with a fixed schedule of random seeds, and Section 5.2 reports the estimated probabilities together with the confidence implied by these parameters. The task configuration used by the system experiments is given in Table 3: three preemptible tasks with the same initial priority, a worst-case execution time of 5 ticks and a period of 30 ticks.

Table 3 Task configuration for the thread management module system experiments

Task name	Task ID	Preemptible	Initial priority	Worst-case execution time	Period
Runner	1	Yes	5	5	30
Task0	2	Yes	5	5	30
Task1	3	Yes	5	5	30

Four metrics are used throughout the evaluation. M1 is property coverage: how many of the target properties of Table 1 are discharged, by which mechanism, and with what evidential strength. M2 is verification cost, measured by the size of the model that has to be analyzed rather than by wall-clock time, since the latter is dominated by the host machine. M3 is consistency: whether the model and the implementation agree on every observable record of every generated scenario, and if not, at which record they first diverge. M4 is regeneration cost: which artifacts have to be edited when a requirement or the kernel changes. M1 and M2 answer RQ1, M3 answers RQ2, and M2 together with M4 answers RQ3. Table 4 maps each research question to its metric, its evidence and the subsection in which the answer is given.

Table 4. Mapping from the research questions to the evidence collected and to the subsection in which each question is answered.

RQ	Metric	Evidence collected	Section	Answer
RQ1	M1, M2	15 target properties of the task-management module, out of 23 properties specified on six modules; per-module model size and the horizon that bounds the state space	5.2	All 15 properties hold; three different discharging mechanisms are used; the module-level state space remains analyzable
RQ2	M3	1,638 abstract execution traces converted into tests; record-by-record comparison of CALL, SCALAR, STATE, SIGNAL and WAIT records on the concrete kernel	5.3	Observable equivalence holds for every generated scenario; a divergence is localized to one record in one segment

RQ	Metric	Evidence collected	Section	Answer
RQ3	M2, M4	Artifact sizes of six modules (Table 6) and the set of artifacts that must be edited after a kernel or requirement change	5.4	Verified modules are reusable and regenerable; the refinement dictionary is the only artifact edited per API change

5.2 RQ1: Are the target properties verifiable, and what do the results mean?

In the following, we take task execution timeliness as an example to illustrate the verification process. The model constructed to verify the task execution timeliness property is Scenario.taskTimeOut.

This property asserts that the total execution time of a task (including CPU execution time, resource waiting time, and scheduling overhead) never exceeds the task period.

We use an auxiliary function task_timeout() to retrieve the state value of taskTimeOut, along with another auxiliary function trace_b("Scenario.taskTimeOut", taskTimeOut) to monitor the taskTimeOut state under arbitrary system behaviors. The internal logic of task_timeout() works as follows: it iterates over all tasks and calculates the total time consumed from task activation to completion. If the total time consumption of any task exceeds its task period, the function returns false; otherwise, it returns true.

All execution traces generated by the model yield a return value of True. This result shows that the estimated probability of the property Scenario.taskTimeOut holding is 100%, demonstrating that system behaviors satisfy the task execution timeliness property.

Table 5. Property-level results for the task management module. The five safety properties are discharged by model checking or by a structural guard; the ten functional properties are discharged by trace-driven equivalence checking.

No.	Target property	Discharged by	Horizon / evidence	Result
1	Deadlock-free task scheduling	structural progress guard of the test-driver state machines (Section 4.2.3)	whole run; terminated run	Holds in all runs
2	At most one task in the running state	G{1000} (Scenario.atMostOneRunning)	1,000 steps	Holds, p = 1.000
3	Task execution timeliness	G{1000} (Scenario.taskTimeOut)	1,000 steps	Holds, p = 1.000
4	Task priority protection	G{1000} (Scenario.prioInversionAvoid)	1,000 steps	Holds, p = 1.000
5	Data access consistency	G{1000} (Scenario.mutexLockHoldingValid)	1,000 steps	Holds, p = 1.000
6	Verification of invalid task creation	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
7	Verification of invalid task deletion	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
8	Verification of valid task creation and deletion	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent

No.	Target property	Discharged by	Horizon / evidence	Result
9	Verification of invalid task activation	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
10	Verification of valid task activation	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
11	Verification of invalid task suspension	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
12	Verification of valid task suspension	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
13	Verification of invalid task resumption	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
14	Verification of valid task resumption	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent
15	Priority lowering for a task holding a mutex	trace replay and record-level equivalence (Section 5.3)	one generated test	Consistent

Table 5 is read in two ways. As a verification report, it states that every property identified from the requirements of the kernel in Table 1 is discharged, and it distinguishes the strength of the evidence: the four temporal properties are guaranteed statistically, over the stated horizon and confidence, whereas the ten functional properties are guaranteed exhaustively with respect to the generated traces. As a specification report, it shows that the property set is not homogeneous and that a single tool cannot discharge all of it, which is why the model combines an observer-plus-LTL mechanism with a structural progress guard.

The meaning of each result for the kernel is worth stating explicitly. At-most-one-running excludes a scheduler that leaves two tasks in the running state, which on a single-core target would corrupt the ready queue and, on a multi-core target, would produce simultaneous access to a shared structure. Timeliness bounds the time from activation to completion, including waiting and scheduling, so a violation is the formal counterpart of a missed deadline. Priority protection and mutex-holding validity together bound priority inversion: the first forbids lowering the priority of a task that holds a lock governed by the protocol, the second forbids two tasks from holding the same mutex, and a failure of either is what turns a bounded inversion into an unbounded one. Deadlock freedom is the weakest of the five in evidential terms, because it is witnessed by termination rather than proved over all executions, and this asymmetry is deliberate: it is the property for which model checking buys the least, since a non-terminating run is directly observed.

Two observations qualify the result. First, a green property set is informative only together with the scenario space that was explored: the model fixes three tasks, one mutex and a fixed-priority preemptive policy, so the properties are established for that configuration and not for arbitrary task sets. Second, the property set is also used in negated form during model development: a formula such as G{1000} (not Scenario.prioInversionAvoid) is not a requirement but an enumerator, because it forces the engine to exhibit scenarios in which the invariant would be broken. Such a trace is either a genuine defect or evidence that the invariant was

mis-formalized, and both outcomes were useful while the model was being built. This is a practical conclusion about the specification exercise: writing a property in negated form is a cheap way to test the property itself.

Finally, the exercise produced a reusable property vocabulary rather than a single result. The same five requirement shapes, namely mutual exclusion of the running state, bounded time-to-completion, protocol-constrained priority, single-holder mutex and progress, recur in the semaphore, message-queue, event, heap and barrier models, so that 23 properties could be specified across six modules with only four temporal forms: G{n} over a state predicate, G{n} over a conjunction of predicates, a bounded F{n} for eventual completion, and the negated form used as an enumerator. The horizon differs by module, 1,000 steps for the task and semaphore models and 500 for the heap, and is reduced to a single step where the predicate only has to hold at the next observation. This pattern-to-module mapping is the part of the work that transfers most directly to another kernel.

5.3 RQ2: Does trace-derived testing establish observable equivalence?

5.3.1 Verification method

Module functional correctness verification treats user interface layer APIs as basic functional units and checks whether their behavior conforms to design expectations. In contrast, functional consistency verification leverages refinement techniques to provide evidence of observable equivalence between a module's concrete implementation and its abstract behavior over the generated traces. We first describe the specific consistency verification method.

As introduced earlier, tagged directives are defined to extract abstract behavior traces from the BIP model. During verification by the BIP engine, BIP behavior traces are exported as text-based abstractions via C standard library printf statements, yielding the abstract behavior traces.

For instance, when task_create is invoked within the abstract model, we capture the corresponding behavior trace, which records whether the API is invoked, the input arguments, and the return code. To achieve this, printf markers are inserted before and after each task_create call.

```
printf("@@@ %d CALL task_create %d %d %d %d createRC\n",
                    bid, name, prio, mode, attr, tsk0_id);
createRC = abs_task_create(bid, tsk0_id, name, prio, preempt, setRC);
printf("@@@ %d SCALAR createRC %d\n", bid, createRC);
```

Each output entry starts with the tag string “@@@”, which enables filtering of these records from the raw output produced by the BIP engine. Every output record carries a unique bid identifier that tracks state transitions among randomly executed BIP components. This information supports the generation of test code capable of reproducing the corresponding scenario.

A sample output for the above example is shown below:

```
@@@ 4 CALL task_create 1 5 0 0 2 createRC
@@@ 4 SCALAR createRC 0
```

It can be observed that BIP component 4 invokes task_create to spawn a task with name = 1, prio = 5, mode = 0, attr = 0, and id = 2. The return code is stored in the variable createRC, which is a scalar holding a current value of zero.

5.3.2 Analysis of verification results

We use an invalid task creation case to illustrate the details of module functional correctness

and consistency verification.

The record types define, by construction, the classes of divergence that the comparison can expose, and it is useful to separate them because they are not equally accessible to conventional testing. Class (i) is ordering: a CALL record appears at a different position in the two streams, which indicates a scheduling or synchronization difference. Class (ii) is argument mismatch: the same action is invoked with different values, which indicates a marshalling or default-value difference between model and kernel. Class (iii) is status mismatch: the SCALAR record carries a different return code, which indicates a disagreement about the validity conditions of the API. Class (iv) is state mismatch: a STATE record reports a different task state after the same call. Class (v) is synchronization mismatch: a SIGNAL or WAIT record is missing, duplicated or inverted. Classes (i) and (v) are the ones that hand-written functional tests rarely expose, because exposing them requires reproducing a particular interleaving, whereas a trace supplies that interleaving by construction.

As shown in Fig. 6, the abstract model layer invokes the abstract behavior task_create to create a task with parameters (0, 5, 0, 0, 2), and the return code createRC equals 3 (marked by the two "@@@" tagged output lines, lines 2 and 5). At the system implementation layer, the user-level interface API is called to create a task with identical input parameters, returning 0x3 (the two lines between the "@@@"-tagged outputs, lines 3 and 4).

```
L:Runner running
L:@@@ 4 CALL task_create 0 5 0 0 2 createRC
L:Creating Task(0, 5, 0, 0, RTEMS_MINIMUM_STACK_SIZE, 2)
L:Returned 0x3 from Create
L:@@@ 4 SCALAR createRC 3
L:Runner finished
```

Fig. 6 Verification results for the functional correctness of invalid task creation.

Both the abstract layer and implementation layer invoke the task creation behavior with identical parameters and yield a consistent return value of 3, indicating that task creation fails due to an invalid task name. This result confirms the functional correctness of both layers and provides evidence of their observable equivalence on this scenario.

Two conclusions follow. First, the invalid-creation case shows that the equivalence is checked at the level of the API contract rather than of the implementation: the model and the kernel agree not only on the fact that the call fails but on the class of the failure, reported as status 3 for an invalid task name, and on the absence of a task-side effect, which is the part of the contract that hand-written tests most often leave unverified. Second, the same comparison applies unchanged to the nine remaining functional properties of Table 5, because the difference between them is an action name and an argument vector, both of which are carried by the CALL record; once the model path for an API exists, covering a further scenario costs no additional verification machinery. The limits are equally clear: only instrumented state is observed, so a divergence in a state that emits no record, for example an internal queue index that no API exposes, is invisible to this mechanism. That gap is the subject of Section 5.5.

5.4 RQ3: What does the workflow cost, and what can be reused?

The cost of the workflow is dominated by the artifacts that have to be maintained, not by the checking itself. Table 6 reports their size for the six modeled modules, counted from the model

and support-layer sources. Two features are worth noting. First, the BIP model is a small part of the artifact set: about 4,500 lines of model are supported by about 8,300 lines of C++ support layer, because the support layer carries the kernel semantics that the model abstracts, and it is written once and reused by every module that shares the same task or semaphore substrate. Second, the size does not grow with the number of scenarios: adding a scenario adds transitions to an existing driver model rather than a new model, which is why the event model, with 11 atomic components, is the largest of the six while remaining of the same order as the others.

Table 6. Size of the modeling artifacts for the six modules, counted from the sources.

Module	.bip model (lines)	Support layer (lines)	Atomic components	Ports	Connectors
Task management	883	1,515	7	90	8
Semaphore	529	1,930	4	47	5
Message queue	454	1,777	4	27	5
Event	1,072	1,163	11	39	16
Heap management	876	1,190	4	35	18
Barrier	723	710	6	38	7
Total	4,537	8,285	36	276	59

Reuse operates at two levels. Within a module, a verified atomic component is reused by every scenario that exercises it: the four test drivers of the task-management model share the same worker-task models, the same system and clock models, and the same observers, so the verification argument for the shared part is made once and inherited by the other scenarios. Across modules, the support layer and the observer functions are shared wherever the same kernel concept appears, and the refinement dictionary is shared by every module whose API surface overlaps. This is the concrete content of the reusability claim: it is not that a model can be moved between kernels unchanged, but that the parts which encode kernel semantics are identified, isolated and therefore maintainable.

Regeneration after a change is correspondingly narrow. If an implementation changes but the requirement and the model do not, only the tests are regenerated and replayed, which is a re-run of the generator and of the test build; if a requirement changes, the affected property and its observer change, and the model is re-checked, while the dictionary is untouched; if an API signature changes, the dictionary template for that action changes, which is the only edit that touches hand-written text. Compared with building a single model of the whole kernel, the module-level organisation pays a fixed cost, namely the interface assumptions that each module records about its neighbours, and buys a state space that stays within reach of statistical checking and a verification argument that can be maintained incrementally. This trade-off, rather than a claim of lower total effort, is what the experiment supports.

5.5 Threats to validity

Internal validity. The abstract model is not the kernel: the support layer re-implements kernel services in C++ so that the model can be executed, and any misunderstanding of the kernel semantics is reproduced faithfully by the model. Two measures bound this risk. The observer functions are written against the same API semantics that the kernel documents, and the refinement step executes on the concrete kernel, so a semantic misunderstanding that changes

observable behavior appears as a divergence rather than as a passing model. A misunderstanding that changes no observable record, however, remains undetected.

Construct validity. The property set is derived from the design requirements of the module, and a property that is not in the set is not verified. The set covers mutual exclusion, bounded time, protocol-constrained priority, single-holder mutex and progress, but it does not cover interrupt behavior, multi-core interference at the hardware level, or absolute worst-case execution time, all of which belong to the timing analysis rather than to the functional model. The horizon of the bounded operators is a second construct-level choice: a property is guaranteed over 500 or 1,000 steps of a run, not over unbounded executions.

External validity. The evaluation covers one proprietary kernel, one scheduling policy and small configurations of three tasks, one mutex and one barrier. The claim that the workflow transfers rests on the structure of the artifacts, namely the separation of model, observer and dictionary, and not on a measurement across kernels; a kernel whose API has substantially different semantics would require a new support layer, although not a new method.

Conclusion validity. A property set that passes must not be read as a statement that the kernel is free of defects. The statistical results are statements about the probability of a violation under the sampled schedule and the stated confidence, and the equivalence results are statements about the traces that were generated, whose coverage is that of the scenario model. The defensible reading is therefore narrower and, for an industrial audience, more useful: within the explored scenario space, the kernel exhibits the specified behavior, and the mechanism that established this continues to hold as the kernel evolves.

6 Conclusions

Formal methods can prove the functional correctness of embedded real-time operating systems, but the overhead of conventional verification has kept them out of industrial RTOS development. This paper has asked whether the overhead can be brought down far enough for verification to survive the release cycle of a kernel, and has answered the question in three parts.

For RQ1, a requirement-level BIP model of a kernel module is tractable: 15 target properties of the task-management module are discharged over models of a few hundred lines, supported by an observer layer that expresses the properties as exported state predicates, and the state space stays analyzable because it is bounded per module and per horizon. The answer is qualified in two ways that matter in practice: the properties are guaranteed statistically rather than exhaustively, and progress properties are better discharged structurally than by temporal logic.

For RQ2, trace-derived testing provides evidence of observable equivalence, and it does so at a granularity that hand-written conformance tests do not reach: because a trace carries the component identifier, the arguments and the effect of every API call, a divergence is localized to one record in one scenario, and the classes of divergence it can expose include ordering and synchronization mismatches. The evidence is exhaustive with respect to the generated traces, and the trace set is in turn limited by the scenario model.

For RQ3, the cost of the workflow is carried by artifacts rather than by proof effort, and that is what makes it maintainable: the model, the observer functions and the refinement dictionary change independently, so a kernel change touches the dictionary, a requirement change touches a property and its observer, and an implementation change touches only the regenerated tests. Verified modules are then reusable across scenarios and across modules that share the same kernel substrate. The practical implication for an embedded RTOS team is that verification can be

introduced module by module, on the critical modules first, and grown without a point at which the whole argument has to be redone.

The work also delimits its own scope. Interrupts, multi-core interference at the hardware level and absolute worst-case execution time are outside the current model, and the reported configurations are small. Future work therefore proceeds along two lines: extending the model to interrupt entry and exit and to symmetric multi-core scheduling, which requires a new support layer rather than a new method, and enriching the property vocabulary with the timing-analysis obligations that the current set leaves to separate tools.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used DeepSeek in order to assist with language editing and the refinement of content. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

References

[1] R. Luna, S.A. Islam, Security and Reliability of Safety-Critical RTOS, SN Comput. Sci. 2 (2021) 356. https://doi.org/10.1007/s42979-021-00753-y.

[2] C.A. Raj, Innovations in Real-Time Operating Systems (RTOS) for Safety-Critical Embedded Systems, J. Comput. Sci. Technol. Stud. 7 (3) (2025) 791 – 797.https://doi.org/10.32996/jcsts.2025.7.3.86.

[3] C. De Sio, S. Azimi, L. Sterpone, Evaluating reliability against SEE of embedded systems: A comparison of RTOS and bare-metal approaches, Microelectron. Reliab. 150 (2023) 115124. https://doi.org/10.1016/j.microrel.2023.115124.

[4] J. Shi, C.-C. von Egidy, K.-H. Chen, J.-J. Chen, Formal Verification of Resource Synchronization Protocol Implementations: A Case Study in RTEMS, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 4157–4168. https://doi.org/10.1109/TCAD.2022.3197501.

[5] J. Yin, X.L. Chen, J. Yan, S. Song, Y. Han, Formal Verification of Partitioned Scheduling in Real-Time Operating Systems Using Timed Automata, J. Softw. Evol. Process (2026). https://doi.org/10.1002/smr.70077.

[6] L.B. Ribeiro, F. Lorber, U. Nyman, K.G. Larsen, M. Baunach, A Modeling Concept for Formal Verification of OS-Based Compositional Software, in: L. Lambers, S. Uchitel (Eds.), Fundamental Approaches to Software Engineering, FASE 2023, Lecture Notes in Computer Science, vol. 13991, Springer, Cham, 2023, pp. 26 – 46. https://doi.org/10.1007/978-3-031-30826-0_2.

[7] G. Klein, Operating System Verification — An Overview, Sadhana 34 (1) (2009) 27 – 69. https://doi.org/10.1007/s12046-009-0002-4.

[8] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, J. Szefer, Survey of Approaches and Techniques for Security Verification of Computer Systems, ACM J. Emerg. Technol. Comput. Syst. 19 (1) (2023) 6. https://doi.org/10.1145/3564785.

[9] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP). 2009: 207-220.

[10] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, D. Costanzo, CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels, in: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association, 2016, pp. 653–669.

[11] L. Gäher, M. Sammler, R. Jung, R. Krebbers, D. Dreyer, RefinedRust: A type system for high-assurance verification of Rust programs, Proc. ACM Program. Lang. 8 (PLDI) (2024) 192. https://doi.org/10.1145/3656422.

[12] Ayoun SÉ, Denis X, Maksimović P, Gardner P. A hybrid approach to semi-automated Rust verification. arXiv:2403.15122, 2024.

[13] H. Afzaal, M. Imran, M.U. Janjua, Formal verification of persistence and liveness in the trust-based blockchain crowdsourcing consensus protocol, Comput. Commun. 192 (2022) 384–401. https://doi.org/10.1016/j.comcom.2022.06.014.

[14] H.N. Feng, J.J. Guan, H. Li, X.S. Pan, Z.M. Zhao, FIDO gets verified: A formal analysis of the universal authentication framework protocol, IEEE Trans. Dependable Secure Comput. 20 (5) (2023) 4291–4310. https://doi.org/10.1109/TDSC.2022.3217259.

[15] X. Chen, L. Qiao, H. Liu, Z. Ma, J. Jiang, Security Verification Method of Embedded Operating System Semaphore Mechanism based on Coq, in: 2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE), 2021, pp. 392–395. https://doi.org/10.1109/ICBASE53849.2021.00079.

[16] N. Chong, B. Jacobs, Formally Verifying FreeRTOS' Interprocess Communication Mechanism, 2021. (Available online).

[17] S. Jiménez Gil, M.I. Capel, G. Olea Olea, Automatic test cases generation from formal contracts, Inf. Softw. Technol. 172 (2024) 107467. https://doi.org/10.1016/j.infsof.2024.107467.

[18] R. Zrelli, H.A. Misson, S. Kamkuimo, M. Ben Attia, A. Shabah, F.G. de Magalhaes, G. Nicolescu, Integrating formal methods and automated tools for DO-178C compliance in UAV software, Inf. Softw. Technol. 194 (2026) 108068. https://doi.org/10.1016/j.infsof.2026.108068.

[19] L. Gu, A. Vaynberg, B. Ford, et al., CertiKOS: A certified kernel for secure cloud computing, in: Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys), 2011.

[20] R. Gu, Z. Shao, J. Kim, et al., Certified concurrent abstraction layers, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2018, pp. 646–661. https://doi.org/10.1145/3192366.3192381.

[21] F. Xu, M. Fu, X. Feng, et al., A practical verification framework for preemptive OS kernels, in: Proceedings of the 28th International Conference on Computer Aided Verification (CAV), 2016, pp. 59–79. https://doi.org/10.1007/978-3-319-41540-6_4.

[22] Y. Zhao, D. Sanán, F. Zhang, et al., Refinement-based specification and security analysis of separation kernels, IEEE Trans. Dependable Secure Comput. 16 (1) (2017) 127–141.

https://doi.org/10.1109/TDSC.2017.2671760.
[23] F. Zhang, L. Zhang, Y. Zhao, et al., Refinement-based Specification and Analysis of Multi-core ARINC 653 Using Event-B, Form. Asp. Comput. 35 (4) (2023) 1 – 29. https://doi.org/10.1145/3599731.
[24] H. Sun, H. Lei, L. Qiao, et al., Formal verification of GP specification based embedded operating system, in: Proceedings of the 2018 International Conference on Computer Science, Engineering and Applications (CSEA), 2018.
[25]] A. Butterfield, F. Tuong, Applying formal verification to an open-source real-time operating system, in: Proceedings of the 2023 International Symposium on Theoretical Aspects of Software Engineering (TASE), 2023.
[26] H. Evrard, A.F. Donaldson, Analysing futex-based synchronisation primitives using model checking, Form. Methods Syst. Des. (2025).
[27] Q. Carbonneaux, N. Zilberstein, C. Klee, et al., Applying formal verification to microkernel IPC at Meta, in: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), 2022, pp. 116 – 129. https://doi.org/10.1145/3497775.3503681.
[28] A. Basu, M. Bozga, J. Sifakis, Modeling Heterogeneous Real-time Components in BIP, in: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM’06), 2006, pp. 3 – 12. https://doi.org/10.1109/SEFM.2006.27.
[29] B.L. Mediouni, A. Nouri, M. Bozga, et al., SBIP 2.0: Statistical Model Checking Stochastic Real-Time Systems, in: Automated Technology for Verification and Analysis – 16th International Symposium (ATVA 2018), 2018, pp. 536 – 542. https://doi.org/10.1007/978-3-030-01090-4_31.
[30] C. Li, Y. Zhang, R. Yao, et al., BIPSec: An End-to-End Formal Verification Framework for AH/ESP in Industrial-Grade IPSec Protocol Stacks, in: 2025 IEEE International Conference on Blockchain Technology and Information Security (ICBCTIS), IEEE, 2025, pp. 1 – 8. https://doi.org/10.1109/ICBCTIS66509.2025.11387724.
[31] C. Li, Y. Zhang, X. Yang, et al., IKEv2-FV: A BIP-Based Formal Verification Framework for the IKEv2 Protocol, in: 2025 2nd International Symposium on AI and Cybersecurity (ISAICS), IEEE, 2025, pp. 1 – 8. https://doi.org/10.1109/ISAICS66888.2025.11350064.
[32] A. Metwally, A. Somov, Designing Future Aerospace Systems: Integrating BIP Formal Verification Into F Prime Framework, IEEE Access 13 (2025) 155952 – 155968. https://doi.org/10.1109/ACCESS.2025.3606362.
[33] S. Bliudze, P. van den Bos, M. Huisman, R. Rubbens, L. Safina, JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java, in: L. Lambers, S. Uchitel (Eds.), Fundamental Approaches to Software Engineering, FASE 2023, Lecture Notes in Computer Science, vol. 13991, Springer, Cham, 2023, pp. 143 – 150. https://doi.org/10.1007/978-3-031-30826-0_8.
[34] Q. Wang, H. Dai, Y. Zhao, M. Zhang, S. Bliudze, Enabling Behaviour Tree Verification via a Translation to BIP, in: D. Marmsoler, M. Sun (Eds.), Formal Aspects of Component Software, FACS 2024, Lecture Notes in Computer Science, vol. 15189, Springer, 2024, pp. 3 – 20. https://doi.org/10.1007/978-3-031-71261-6_1.
[35] P.W. O’Hearn, Continuous Reasoning: Scaling the impact of formal methods, in: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science

(LICS), ACM, 2018, pp. 13 – 25. https://doi.org/10.1145/3209108.3209109.