IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received March 23, 2021, accepted April 7, 2021, date of publication April 15, 2021, date of current version April 23, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3073398

Formal Verification of a Hybrid IoT Operating System Model

YUQIAN GUAN[1], JIAN GUO[2], AND QIN LI[3]
[1]Soft/Hardware Co-Design Engineering Research Center, East China Normal University, Shanghai 200062, China
[2]National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, Shanghai 200062, China
[3]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China

Corresponding author: Jian Guo (jguo@sei.ecnu.edu.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2019YFB2102600, and in part by the Project of Science and Technology Commitment of Shanghai under Grant 19511103602.

ABSTRACT The Internet of Things (IoT) is becoming an increasingly common paradigm. As IoT usage scenarios have increased, many challenges in IoT operating systems' safety and adaptability have remained. According to the programming model, IoT operating systems can be categorized into three types: multithreading, event-driven, and hybrid. Different operating system models are applied in different scenarios depending on the real-time requirements or resource richness. The safety of IoT operating systems is critical; hence, formal verification is an important method of detecting potential vulnerabilities and providing safety guarantees. This paper proposes a hybrid model for an IoT operating system and employs the Event-B method for modeling and verification. We rewrite the requirements and divide the Event-Bus hybrid operating system model into eight levels for refinement. The safety and liveness properties of Event-Bus are guaranteed by generating and proving the proof obligations at each model level. A large proportion of the proof obligations (91%) are automatically proven on the Rodin platform to simplify the development process.

INDEX TERMS Formal verification, operating systems, formal specifications, event-B.

I. INTRODUCTION

With the improvement of network communication technology and the decrease in the cost of hardware computing power, the era of the Internet of Things (IoT) has arrived. An IoT operating system (OS) provides basic functions (including task management, memory management, and communication management) under limited resources. Due to various scenario limitations, IoT devices are often required to complete specified computations using small volumes and few resources. Consequently, IoT devices need a customized OS.

The programming models for IoT OSs can be classified into three types [1]: multithreading, event-driven, and hybrid. The task of multithreading OSs (such as Lite OS [2] and RIOT [3]) is called a thread. In multithreading OSs, the running resources of the lower-priority thread are preempted by the higher-priority thread. The disadvantages of this approach are the large memory space required to save each thread's context and the time consumed in switching threads. In an event-driven OS, such as the Oberon System [4] or Nano-RK [5], the event handler schedules tasks. An event is triggered when a task finishes, and the event handler executes the next task. The hybrid OS approach was proposed with the development of IoT OSs that combine the event-driven and multithreading programming models. Although many hybrid programming models have been presented to date, these works are subject to various problems, such as poor parallel performance and insufficient scheduling flexibility. Thus, at present, there is still room for optimization in the design of hybrid programming models.

With the development of formal verification technology, the corresponding tools have become increasingly mature. An increasing number of software programs employ formal verification technology, and the importance of formal verification is generally recognized. In formal verification, the validator constructs a model with mathematically precise semantics under design and performs extensive analysis concerning correctness requirements [6].

Event-B [7] is a formal modeling and verification method based on set theory and predicate logic. Event-B adapts refinement theory and a level-by-level method to increase the specification to gradually complete system modeling. The proof obligation [8] provides mathematical proof of the properties according to a set of rules. The Rodin platform

The associate editor coordinating the review of this manuscript and approving it for publication was Zhaojun Li.

[9] introduces support for automatic Event-B model creation and proving the proof obligation. This technique makes heavy usage of events defined utilizing guards and parallel actions. Although very powerful for formal verification proofs, this approach is not entirely satisfactory for developing classic programs, where the dynamic part is defined by operations related to preconditions and sequential actions. We use events to develop a classic approach to address preconditions and sequential action operations in the last refinement model.

This paper contributes to exploring safe and reliable IoT OSs. The major contributions of this work can be summarized as follows:

- A hybrid model of an IoT OS, called Event-Bus, is proposed. Event-Bus consolidates the multithreading model with the publish-subscribe pattern (a particular event-driven model). The main advantage of this approach is the ability to be flexibly configured to adapt to different scenarios.
- The Event-B method is employed to construct Event-Bus's formal model. We prove the safety and liveness properties of Event-Bus in each level of the model using the Rodin platform. The automatic verification rate of the entire verification reaches 91% through the clear model description and the design of an excellent refinement strategy.

The overall structure of this paper has seven sections, including this introduction. Section II introduces the related work of the formal verification of the IoT. Section III gives the background on hybrid IoT OS models and describes the formal modeling and analysis methods. Section IV is concerned with rewritten requirements and proposing a refinement strategy. Section V and Section VI present the process for building the abstracted model and refining each level of the model. Section VII verifies and validates the system. Finally, Section VIII contains the conclusion and ideas for further work.

II. RELATED WORK

In recent years, an increasing amount of literature on IoT environment safety has been published. Formal verification is an important method to ensure the correctness of the system. Approaches for verifying IoT OSs and embedded software can be divided into three categories: (1) verification of application programs with a highly abstracted scheduling policy, (2) verification approaches for the correctness of either OS models or implementations, and (3) verification of monolithic kernels or partial kernel services.

A. VERIFICATION OF APPLICATION PROGRAMS WITH A HIGHLY ABSTRACTED SCHEDULING POLICY

Many studies have shown that the first method can be used effectively to verify embedded software under specific limitations [10]–[12]. Gallardo *et al.* describe an approach to verify concurrent C code by automatically extracting a high-level formal model suitable for analysis with the Construction and Analysis of Distributed Processes (CADP) toolbox [10]. The approach fits well within the existing architecture of CADP, which does not need to be altered to enable C program verification. Inverso *et al.* propose a new approach to Bounded Model Checking (BMC) for sequentially consistent C programs using POSIX threads [11] in which a multithreaded C program is first translated into a nondeterministic sequential C program. This program preserves reachability for all round-robin schedules with a given bound on the number of rounds and then re-uses existing high-performance BMC tools as backends for the sequential verification problem. Rabinovitz *et al.* propose a SAT-based bounded verification technique, called Threaded-C Bounded Model Checking (TCBMC), for threaded C programs [12]. This approach is based on C Bounded Model Checking (CBMC), which models sequential C programs in which the number of executions for each loop and the depth of recursion are bounded. Moreover, bugs that invalidate safety properties can be detected. These include races and deadlocks, the detection of which is crucial for concurrent programs. This category of approaches assumes arbitrary interleavings among tasks but reduces the verification complexity by either using partial-order reduction, limiting the number of context switches among tasks, which suffer from a high false-alarm rate and additional costs for refinements.

B. VERIFICATION APPROACHES FOR THE CORRECTNESS OF EITHER OS MODELS OR IMPLEMENTATIONS

Several approaches realize the importance of addressing OS-related issues and compose formal OS models by converting C programs into the modeling language used to model the OS [13]–[15]. Klein *et al.* transform the seL4 microkernel into a Haskell model for formal development [13]. In this case, the correctness of the compiler, assembly code, and hardware is assumed. They use a unique design approach that fuses formal and OS techniques. The implementation strictly follows a high-level abstract specification of kernel behavior that encompasses both traditional design and implementation safety properties. Thus, the kernel will never crash and will never perform an unsafe operation. There are many formal verifications of the OSEK/VDX OS in the context of IoT OSs. Huang *et al.* employ the process algebra named Communicating Sequential Processes (CSP) to describe and analyze a real code-level OSEK/VDX OS, and they formally model the whole system as a CSP process encoded and implemented in the process analysis toolkit [14]. The expected properties are described and expressed in terms of the first-order logic. Zhu *et al.* present a unified executable formal automobile kernel under the OSEK/VDX standard by defining the system services' operational semantics in the standard using a rewrite-based executable semantic framework called K [15]. They identify several ambiguities in the OSEK/VDX standard and a potential deadlock vulnerability in an industrial automobile application. However, this category of methods suffers from high verification costs due to comprehensive interpretation of the program code and the formal OS model.

C. VERIFICATION OF MONOLITHIC KERNELS OR PARTIAL KERNEL SERVICES

The last approach is to verify only the unexpanded kernel or partial kernel services. By reducing the scope of formal verification, this method simplifies formal development and allows an IoT system's vital properties to be verified. Software verification has many successful applications regarding monolithic OS kernels and their extensions, including verification of OS device drivers [16], [17], verification of network protocols [18], [19], and verification of a memory management subsystem [20], [21]. Because of the rich expressiveness of the Event-B method, many studies use Event-B to verify the kernel services of an IoT OS. Amjad *et al.* use the Event-B formal verification method to model and verify the ZigBee protocol stack by embedding the protocol primitives in Event-B [22]. Wen *et al.* present the formal development of a memory management module for a real-time OS [23]. They propose various novel techniques in relation to a modular software structure, corresponding code generation, and related Event-B approaches.

As IoT OS features become increasingly abundant, the supported programming model also changes from the original single multithreaded or event-driven model to a hybrid programming model. Nevertheless, the formal verification of hybrid OS models is an underexplored domain.

III. BACKGROUND

A hybrid OS model is a combination of an event-driven model and a multithreading model. Event-B is an event-driven formal verification method in which an event-driven model can be easily constructed. Hybrid OS models and the Event-B method are introduced in this section.

A. HYBRID OS MODELS

IoT scenarios often have real-time requirements or few memory resources. There are scenarios where the OS's real-time requirements are rigorous: a quintessential example is the Internet of Vehicles (IoV). Multithreading OSs, which typically employ preemptive scheduling, are suitable for IoV applications. Tasks with higher priority can be executed faster, and tasks with the same priority can be performed in turn to simulate concurrent effects. The guarantee of real-time performance and the realization of parallel scenarios are among the advantages of the multithreading OS [24]. However, its main disadvantage is considerable computing and memory resource consumption. In scarce resource scenarios, for example, wireless sensor networks (WSNs), more attention is given to controlling the power consumption and memory resource consumption of the OS [25]. Many WSN OSs' programming models are event-driven. Event-driven OS scheduling is mostly cooperative, avoids frequent context switching, and saves separate stack space for each task. Tasks are executed efficiently and require few resources [26] in event-driven OSs. However, the monopolization of computing resources by long tasks is a problem in event-driven OSs that makes the system's real-time performance unsatisfactory [27].

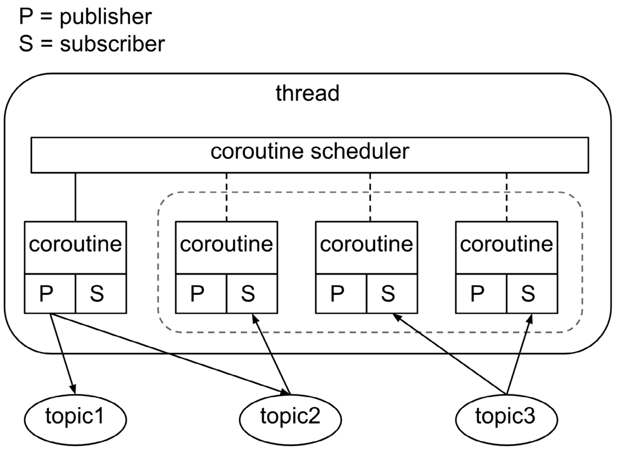


FIGURE 1. The relationship between objects in the Event-Bus hybrid OS model.

A hybrid OS, such as OpenSwarm [28] or SenSpire OS [29], combines the approaches of multithreading and event-driven programming models and therefore offers flexibility in expressing and customizing different scheduling policies. The previously proposed hybrid programming models are designed for specific application scenarios, and they face problems such as inflexible scheduling strategies and poor concurrent performance. OpenSwarm has two modules, an event manager and a thread manager, which schedule event handlers and threads, respectively. An event handler in the event manager can be scheduled only in accordance with the first-in first-out (FIFO) strategy. Both the event handlers and threads in SenSpire OS have priority. In addition to a time-sharing thread scheduler and a FIFO event scheduler, it also has a priority-based event scheduler. The hybrid models of SenSpire OS and OpenSwarm do not consider the optimization of concurrent performance because the functions they support are sufficient to cope with their application environments. However, the IoT application scope is becoming increasingly wider, and the corresponding scenarios are becoming increasingly complex. Hence, hybrid models have room for further optimization in terms of flexibility and concurrency.

Event-Bus is a hybrid programming OS that uses concurrent coroutines combined with a subscription publishing model to implement a hybrid model. The publish-subscribe pattern is a special type of event-driven model. In the Event-Bus OS, each thread contains multiple coroutines. A coroutine is a general-purpose subroutine for non-preemptive multitasking, for which execution can be suspended and resumed. At most one coroutine can run simultaneously in each thread. Concurrent coroutines have two features: 1) the same coroutine can run on different threads, and 2) the same piece of code can generate multiple coroutines, which can be executed in parallel. In the schematic presented in Figure 1, the coroutine outside the dashed box is the running coroutine, and the remaining coroutines are in the waiting queue. There are many topics in such an OS. A coroutine can be a subscriber (S in the figure), with the purpose of subscribing to related topics, or a publisher (P in the figure), which the purpose of publishing events to a topic. Each coroutine can publish messages or subscribe to topics using the publish-subscribe pattern.

Event-Bus also has the advantages of this hybrid programming model, namely, the efficiency of the event-driven approach and the expressiveness of threads. In Event-Bus, the applications represented in the multithreaded model are extensible, and collaboration services are executed in the event-driven model to save resources and achieve high performance. In addition, Event-Bus has several notable characteristics by virtue of its design:

- **Integrated communication between modules:** Event-Bus supports two programming models and realizes synchronous and asynchronous communication functions between modules. It overcomes the problems of synchronization and execution order caused by concurrency by means of the built-in communication mechanism between modules.
- **Support for complex and customized event scheduling methods:** In the Event-Bus model, each thread is an event scheduler, allowing different event handles to respond simultaneously on a multicore platform. Thus, event-driven programs can be more quickly responded to. Developers can process a single sequence of programs with a single-thread and single-coroutine structure, use a single-thread and multiple-coroutine structure to write collaborative services, or design multiple threads and multiple coroutines to achieve concurrent programming.
- **Simplification of event-driven programs:** In programs implemented on the basis of traditional event-driven models, event handles are distributed among different functions, and it is impossible to directly understand the relationship between events from the code. Moreover, additional design is required to ensure the order of code execution. Event-Bus uses coroutines to solve these problems that that can easily complicate the program logic in event-driven systems, thereby simplifying the coding logic of programs.

The combination of coroutines and the publish-subscribe pattern gives the Event-Bus model advantages in terms of concurrent processing and customized scheduling. However, its correctness and safety need to be further verified using formal methods.

B. EVENT-B METHOD

Event-B is a formal method for discrete system modeling based on the B-Method and developed from the idea of action systems [30], [31]. The obligation of proof characterizes the semantics of the Event-B model. Event-B focuses on proof obligation: it expresses the system at different levels

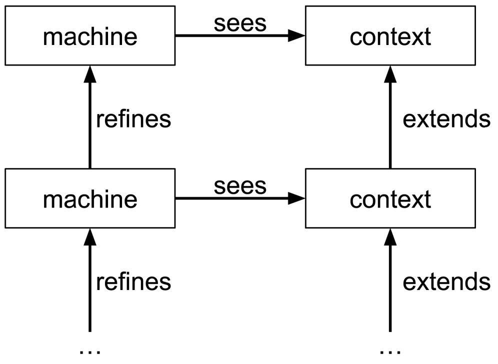


FIGURE 2. The relationship between the abstract machine and context file in the Event-B method.

through refinement [32] and uses mathematical proofs to ensure consistency between models. The methodology is customized with respect to system requirements in different projects. Event-B is an event-driven formal verification method whose operation can be seen as the continuous execution of events that meet the conditions, making it easier to model event-driven behavior and concurrent processes. The multithreading component of the Event-Bus hybrid OS model involves cooperative serialization operations. We define the "running state" enumerator, thereby transforming guarded events into preconditioned operations. This technology [23] enables parallel events in Event-B to run in sequence according to the requirements.

The Event-B model consists of two parts: dynamic and static. The static part of the model is described in the context file, which includes the vectors' set, constants, axioms, etc. The model's dynamic part is implemented by the abstract machine and includes variables, invariants, theorems, events, etc. Their relationship is briefly summarized as follows: machines see contexts, contexts can be extended, machines can be refined. The relationship between the abstract machine and context in the Event-B method is shown in Figure 2.

This work relies on a formal verification and development framework based on the Event-B method [7]. We adopted the Event-B method and the Rodin platform to analyze the Event-Bus hybrid model. The method's framework is shown in Figure 3. In this framework, the requirements are rewritten to make the description more accurate [33]. The rewritten requirements are carefully taken into account and summarized following top-down vertical refinement. A mapping between requirements and refinement is designed to obtain a refinement strategy and confirm that the refinement strategy is satisfactory. According to the refinement strategy, requirements are converted into events and actions to complete the model's vertical refinement. We add attributes or refine operations through a series of model refinements, complete the model step by step, and finally simulate the entire system. Invariants and theorems describing properties in the model can be converted into corresponding proof obligations based on rules. The correctness properties of the model are guaranteed by fully proving the related proof obligations.

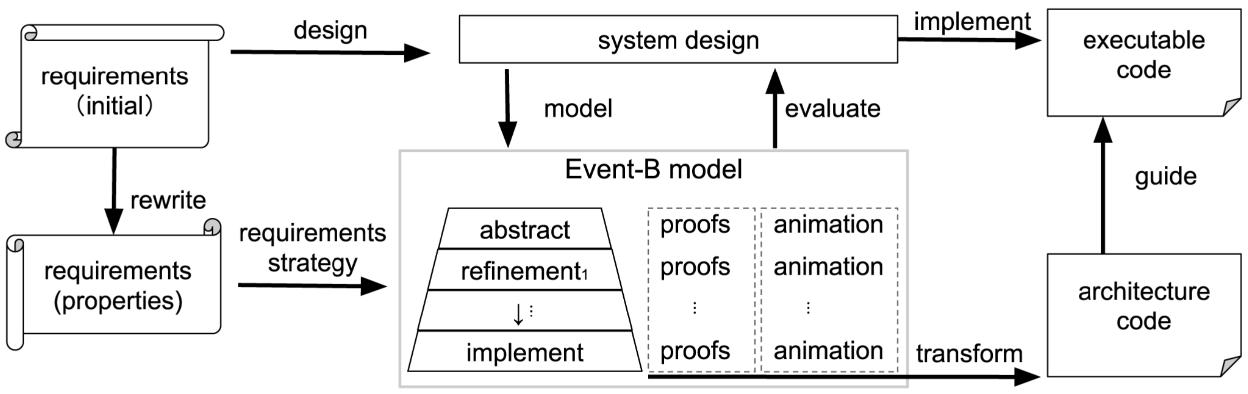


FIGURE 3. **Formal verification and development framework based on the Event-B method.**

TABLE 1. **Basic mathematical notations used for Event-B in this paper.**

Notation	Meaning
$\exists$	There exists
$\forall$	For all
$\in$	Element of
$\notin$	Not an element of
$\vee$	Logical disjunction
$\wedge$	Logical conjunction
$\Rightarrow$	Logical implication
$\mapsto$	Maplet
$\rightarrow$	Total function
$\leftrightarrow$	Relation
$\mathbb{P}$	The set of all subsets (power set)
$\mathbb{N}_1$	The set of positive natural numbers
$:=$	Becomes equal to
$\varnothing$	Empty set
$\cup$	Union
$\bigcup$	N-ary union
$\setminus$	Set minus
$=$	Equal to
$\neq$	Not equal to
$..$	Upto sign
$\triangleright$	Range restriction
$\mid$	Such that

The properties that are already proved to hold in the early models are guaranteed to hold in the later models using refinement. This framework is suitable for verifying a system in the design phase, and the verification result is an assessment of the safety and feasibility of the system design. Moreover, Event-B's implementation model can be converted into architecture code to guide the writing of system executable code.

The basic mathematical notations used for Event-B in this paper are defined in Table 1.

IV. REQUIREMENTS AND REFINEMENT STRATEGY

The requirements are analyzed and rewritten, and a refinement strategy is designed before modeling. An abstract version of the system is built initially. This model reflects the fundamental properties of the system. We gradually refine the model to make it more concrete via a refinement strategy [32]. In this section, we focus on the requirement rewriting analysis and refinement strategy formulation before proceeding to modeling.

A. REQUIREMENTS ANALYSIS

When developers implement the Event-Bus hybrid OS model, they must draw up a series of documents to establish development goals, called initial requirements. In formal development, the initial requirements are an essential reference for modeling. Nevertheless, the system cannot be modeled directly based on these requirements because the initial requirements regularly describe various implementation details with natural language and lack information about the system's operating environment and properties. Therefore the initial requirements are rewritten based on the specification of the Event-B method. We communicate with developers repeatedly based on the initial requirements to determine the correctness and completeness of the rewritten requirements. The initial requirement rewriting process can be divided into three steps:

1) MODIFY

There are some unclear or unreasonable requirements in initial requirements, such as pseudo-implementation descriptions and the pseudo-solution of a problem that is not stated. For the former, we employ it as a comment instead of keeping it in the rewritten requirements. For the latter, we make precise changes through communication with developers.

2) ADD

The number of requirements will increase in two cases. One is to decompose complex requirements. Complex requirements containing multiple pieces can be decomposed to facilitate our subsequent modeling implementation. The other is to increase the correctness requirements. The correctness requirements of the system are essential for verification. Such requirements are often ignored in the development stage. We need to summarize and add to these requirements.

TABLE 2. Refinement strategy of the Event-Bus hybrid model.

Name	Context	Machine	Requirements	Description
Model 0	C0	M0	1 to 8	Build the publish-subscribe pattern model.
Model 1	C0	M1	9 to 14	Implement the message processing method.
Model 2	C1	M2	15 to 19	Introduce the concept of threads.
Model 3	C2	M3	20 to 24	Introduce the concept of coroutines.
Model 4	C2	M4	25 to 26	Decompose the message creation process.
Model 5	C2	M5	27	Replacement model parameters.
Model 6	C3	M6	28 to 35	Introduce the coroutine status.
Model 7	C3	M7	36	Verify the deadlock-free property.

3) CLASSIFY

Requirements describe the system's function, environment, and attributes through natural language, not a pseudo-realization of the system. Their roles are to provide a reference to the correctness of the final constructed system. Requirements are segmented into three categories: **FUN** (functional requirements), **ENV** (environmental requirements), and **SAF** (property requirements).

Event-Bus's rewritten requirements include 36 specifications, consisting of 18 functional requirements, 16 environmental overtures, and 2 property requirements.

B. REFINEMENT STRATEGY

A complex system has numerous requirements that are challenging to fully implement in a single model. The Event-B method solves this problem via a layered and refined approach. First, we build the initial abstract model. Then, corresponding requirements are implemented through multiple refinements of this model. In the refinement process, the model increases in detail and becomes closer to the final implementation. The rewritten requirements are gradually transformed into a modeling language. Given the functions described by various requirements and the dependencies between these functions, we design the refinement strategy to build hierarchically accurate models.

Considering the rewritten requirements, we separate the refinement strategy into 8 levels of models. In Table 2, the rewritten requirements implemented in the model at each level are provided. Each model comprises a context file and an abstract machine, denoted by Cx and My, respectively. The first model is called the abstract model, and the last is the implementation model.

The refinement strategy is divided into the following four stages:

1) THE PUBLISH-SUBSCRIBE PATTERN

Model 0, called the abstract model, models the publish-subscribe pattern, which is the fundamental basis of the Event-Bus model. This model describes the basic function of the publish-subscribe pattern.

2) MESSAGE PROCESSING

In model 1, the concrete realization of message processing is expressed. We add the concepts of the release queue, dispatch queue, and receiving queue. The judgment and handling of illegal and legal messages are distinguished here.

3) THREADS AND COROUTINES

Threads and coroutines are implemented in model 2 to model 5. The two roles of publishers and subscribers are gradually replaced by threads and coroutines.

4) RUNNING STATE

In model 6, the running state is introduced, and the scheduling of the system is modeled.

Model 7 is the last level of the Event-Bus system, and its role is to assist in the proof of the system's deadlock-free property. It does not involve any refinement aspect, so it is not included in the refinement strategy stage.

V. ABSTRACT MODEL

We present the detailed process of building the abstract model in this section. The abstract model is model 0 of the system, and the most abstract concept of the Event-Bus is the publish-subscribe pattern.

A. SPECIFICATIONS

The following are the rewritten requirements in the abstract model. The first four are environmental requirements, explaining the objects in the publish-subscribe pattern and their connections. The next two describing the behavior of the system execution are functional requirements. The remaining two are the property requirements of the system.

The system contains some publishers who can send some messages.	ENV-1
The system contains subscribers who receive some messages.	ENV-2
The system defines a set of topics.	ENV-3
Each message the publisher sends is associated with a unique topic.	ENV-4
Each subscriber can subscribe to topics and add topics to its subscription collection.	FUN-1
Each subscriber can unsubscribe from a topic of interest by removing the topic from its subscription collection. After that, the subscriber will not receive any messages related to the topic.	FUN-2
When a publisher sends a message, subscribers interested in the message's topic will eventually receive the message.	SAF-1
The system is deadlock-free.	SAF-2

B. CONTEXT FILE

The first context file in the abstract model is **C0**, which has four carrier collections that correspond to the sets of topics, subscribers, publishers, and messages. The domains of variables related to these four types of data are *TOP*, *PUB*, *SUB*, and *MSG*. The code is as follows.

```
CONTEXT C0
 SETS
     TOP      \\Topic Collection
     PUB      \\Publisher Collection
     SUB      \\Subscriber Collection
     MSG      \\Message Collection
 END
```

C. INVARIANTS

The abstract model defines 11 invariants to express its rewritten requirements. The first four invariants represent instances of the topic, subscriber, publisher, and message, and they all belong to the corresponding constant set. Invariant **inv1_5** records which topic each message belongs to. Invariant **inv1_6** records the publisher of each message. Invariant **inv1_7** records all messages sent to subscribers. Invariant **inv1_8** describes the variable *erm*, which is related to the time meta variable *era*. The definition of variable *era* is given in invariant **inv1_9**, which is a natural number representing the system time. Variable *stp* records the topics of interest to each subscriber in each period, described in invariant **inv1_3**. Invariant **main_1**, called the "main property" describing the system's liveness, is proposed based on **SAF-1**. We analyze this invariant in detail in Section VII. The variable *erm* represents the relationship between the message and the system time. The definitions of these invariants are as follows.

inv1_1: $top \subseteq TOP$
inv1_2: $sub \subseteq SUB$
inv1_3: $pub \subseteq PUB$
inv1_4: $msg \subseteq MSG$
inv1_5: $tms \in msg \to top$
inv1_6: $pms \in msg \to pub$
inv1_7: $rms \in sub \to \mathbb{P}(msg)$
inv1_8: $erm \in msg \to 1 \,..\, era$
inv1_9: $era \in \mathbb{N}_1$
inv1_10: $stp \in (1 \,..\, era) \to (top \leftrightarrow sub)$
main_1: $\forall s, m \cdot s \in sub \land m \in rms(s) \Rightarrow$
$\quad tms(m) \mapsto s \in stp(erm(m))$

D. EVENTS

In the abstract model, publishers, subscribers, and topics are the three most basic objects. *create_pub*, *create_sub*, and *create_top* are events for creating three objects: publisher, subscriber, and topic, respectively. The code is shown below.

```
create_pub:
  ANY p
  WHERE
     grd1: $p \notin pub$
  THEN
     act1: $pub := pub \cup \{p\}$
  END
```

```
create_sub:
  ANY s
  WHERE
     grd1: $s \notin sub$
  THEN
     act1: $sub := sub \cup \{s\}$
     act2: $rms(s) := \varnothing$
  END
```

```
create_top:
  ANY t
  WHERE
     grd1: $t \notin top$
  THEN
     act1: $top := top \cup \{t\}$
  END
```

Subscribers can be interested in multiple topics. *subscrib_topic* and *rmv_topic* denote the events of subscribing to a topic and unsubscribing from a topic, respectively. The relationship between subscribers and topics is maintained by modifying the variable *stp*. The code is shown below.

```
subscrib_topic:
  ANY s, t
  WHERE
     grd1: $s \in sub$
     grd2: $t \in top$
  THEN
     act1: $stp(era) := stp(era) \cup \{t \mapsto s\}$
  END
```

```
rmv_topic:
  ANY s, t
  WHERE
     grd1: $t \mapsto s \in stp(era)$
  THEN
     act1: $stp := stp \cup$
           $\{era + 1 \mapsto (stp(era) \setminus \{t \mapsto s\})\}$
     act2: $era := era + 1$
  END
```

The event *send_msg* indicates that publisher p publishes message m with topic t. The publisher is not interested in the recipients of the message but cares only about the topic.

```
send_msg:
  ANY p, m, t
  WHERE
    grd1: $p \in pub$
    grd2: $m \notin msg$
    grd3: $t \in top$
  THEN
    act1: $msg := msg \cup \{m\}$
    act2: $tms(m) := t$
    act3: $pms(m) := p$
    act4: $erm(m) := era$
  END
```

The event *handle_msg* is used to detect a newly published message and assign it to each subscriber interested in this message's topic. The subscriber handles all messages sent to it, but the details of message processing are not defined.

```
handle_msg:
  ANY s, m
  WHERE
    grd1: $s \in sub$
    grd2: $m \in msg$
    grd3: $tms(m) \mapsto s \in stp(erm(m))$
    grd4: $m \notin rms(s)$
  THEN
    act1: $rms(s) := rms(s) \cup \{m\}$
  END
```

The refinement process is illustrated after the abstract model. According to the refinement strategy, we gradually build the model by making it increasingly precise such that it expresses more relevant properties.

VI. REFINEMENTS

Based on the refinement strategy, three functional stages are considered: message processing, thread and coroutine, and running state.

A. MESSAGE PROCESSING

The details of message processing are omitted from the abstract model: we describe the messaging strategy in this subsection. The model at this stage corresponds to model 1. The following three rewritten requirements describe the message's delivery and the processing after the message is received.

When a publisher sends a message, multiple copies of the message are sequentially delivered to the queues of subscribers interested in the message's topic.	FUN-3
A message in a subscriber's receiving queue will be accepted by the subscriber and removed from the receiving queue if the topic is still of interest to the subscriber.	FUN-4
A message in a subscriber's receiving queue will not be accepted by the subscriber and will be removed from the receiving queue if the topic is not of interest to the subscriber.	FUN-5

The variable *nyd* represents the message delivery list, shown as invariant **inv2_1**, defined to record the source and destination of each sent message. Invariant **inv2_2** defines the receiving queue *rqu* that is used to store the subscribers' messages. Invariant **inv2_3** introduces the characteristics that every published message is recorded in *nyd*.

inv2_1: $nyd \in msg \leftrightarrow sub$
inv2_2: $rqu \in msg \leftrightarrow sub$
inv2_3: $\forall m \cdot m \in dom(nyd) \Rightarrow nyd[\{m\}] \subseteq stp(erm(m))[\{tms(m)\}]$

The events of the abstract model are modified based on the rewritten requirements. The process of message handling is refined into receiving messages and refusing messages, corresponding to **FUN-2** and **FUN-3**, respectively. The event *handle_msg* in the abstract model is decomposed into two events: *recv_msg* and *refu_msg*.

Two guard conditions are applied during the receive message event. One is message m in the receiving queue of subscriber s, and the other is s still interested in the topic of m. A two-step action is executed if the guard conditions are met. First, m is deleted from s's receiving queue *rqu*. Then, m is accepted by adding m to s's received collection *rms*.

```
recv_msg:
  ANY s, m
  WHERE
    grd1: $m \mapsto s \in rqu$
    grd2: $tms(m) \mapsto s \in stp(era)$
  THEN
    act1: $rqu := rqu \setminus \{m \mapsto s\}$
    act2: $rms(s) := rms(s) \cup \{m\}$
  END
```

The different guard condition for refusing messages is that the current subscriber does not subscribe to a given topic. The action is to remove the message from s's receiving queue *rqu*.

```
refu_msg:
  ANY s, m
  WHERE
    grd1: $m \mapsto s \in rqu$
    grd2: $tms(m) \mapsto s \notin stp(era)$
  THEN
    act1: $rqu := rqu \setminus \{m \mapsto s\}$
  END
```

B. THREADS AND COROUTINES

This section introduces threads and coroutines to replace publishers and subscribers. The refinement of model 2 to model 5 is described this subsection. The rewritten requirements **ENV-5** and **ENV-6** describe the relationship between these objects. We apply various variables and invariants to define threads and coroutines based on Event-B's parallel event feature.

Each thread contains some coroutines.	ENV-5
Each coroutine corresponds to a publisher and a subscriber.	ENV-6

The publisher and the subscriber are not the objects expected by the implementation model but represent the fundamental concepts of Event-Bus. The refinement and replacement of variables can be solved in these refined models. The concepts threads and coroutines replace publishers and subscribers: there are one or more threads in the OS and one or more coroutines in a thread. Each coroutine in Event-Bus can be regarded as being coupled with a publisher and a subscriber. Consequently, each coroutine can send or receive messages and subscribe to topics.

The replacement relationship between the thread/coroutine and the publisher/subscriber is now defined. A series of related variables, such as *rqu* and *nyd*, represents threads, coroutines, and other refined characteristics. The invariants that define these constraints are as follows, where invariant **inv3_1** means variable *th_rq* is the union of the receiving queues of all subscribers in thread *th*. Invariants **inv3_2** and **inv3_3** express the conversion of the message delivery list *nyd* after the introduction of threads. Invariants **inv3_4** and **inv3_5** represent the relationship when the message delivery list *nyd*1 is converted from threads to coroutines. Invariant **inv3_6** enforces the invariance of *cstp*, that is, the coroutine's topic subscription. Invariant **inv3_7** means that *cstp* is derived from the original subscriber- and topic-associated variable *stp*.

```
**inv3_1:** $th\_rq = (\lambda th \cdot th \in thr|$
        $(\bigcup s \cdot s \in ths^{-1}[\{th\}]|rqu \rhd \{s\}))$
**inv3_2:** $nyd1 \in thr \rightarrow (msg \leftrightarrow sub)$
**inv3_3:** $nyd = (\bigcup t \cdot t \in thr|nyd1(t))$
**inv3_4:** $nyd2 \in thr \rightarrow (msg \leftrightarrow crt)$
**inv3_5:** $\forall t, m, s, c \cdot t \in thr \wedge m \in msg$
        $\wedge\ s \in sub \wedge c \in crt \Rightarrow ((s \mapsto c) \in sc \Rightarrow$
        $((m \mapsto s) \in nyd1(t) \Leftrightarrow$
        $(m \mapsto c) \in nyd2(t)))$
**inv3_6:** $cstp \in (1 \,..\, era) \rightarrow (top \leftrightarrow crt)$
**inv3_7:** $\forall e, t, s, c \cdot e \in 1 \,..\, era$
        $\wedge\ t \in top \wedge s \in sub \wedge c \in crt$
        $\Rightarrow$
        $((s \mapsto c) \in sc \Rightarrow$
        $((t \mapsto s) \in stp(e) \Leftrightarrow (t \mapsto c) \in cstp(e)))$
```

Threads and coroutines replace the publishers and subscribers in all events' input parameters. The **WITH** area defines the variable substitution relationship. All variables are related to the publisher or the subscriber, for example, *req* and *nyd* are replaced with the corresponding variables *th_rq* and *nyd*2.

As shown in the replacement in the event *recv_msg*, the input parameter changes from subscriber *s* to coroutine *c*.

```
**recv_msg:**
  **ANY** c, m
  **WHERE**
      grd1: $c \in crt$
      grd2: $crs(c) = TRUE$
      grd3: $m \mapsto sc^{-1}(c) \in th\_rq(ct(c))$
      grd4: false$tms(m) \mapsto c \in cstp(era)$
  **WITH**
      s: $s = sc^{-1}(c)$
  **THEN**
      act1: $cst(c) := sleeping$
      act2: $th\_rq(ct(c)) := th\_rq(ct(c)) \setminus$
            $\{m \mapsto sc^{-1}(c)\}$
      act3: $crms(c) := crms(c) \cup \{m\}$
  **END**
```

C. RUNNING STATE

The last stage consists of model 6, which is the implementation model. The states of a coroutine are illustrated in the implementation model. In an OS, a thread regularly has three states: ready, running, and blocked. The three states of a thread are condensed into two states of a coroutine in the model: active and sleeping (the ready state and running state are combined into the active state). **ENV-7** describes the running states of a coroutine, and **ENV-8** shows that all coroutines in the same thread must be executed sequentially.

The coroutine has two states, active and sleeping, and it is in only one state at a time.	ENV-7
At most one coroutine in a thread is running.	ENV-8

A state set *STT* is added to the context file **C3**, as follows. There are two constants, *active* and *sleeping*, which represent the two states of the coroutine in the model. Axiom **axm1** explains the relationships between the following variables.

```
**CONTEXT** $C3$
  **SETS**
        STT
  **CONSTANTS**
        active, sleeping
  **AXIOMS**
        **axm1:** partition(STT, active, sleeping)
  **END**
```

Variable cst means that each coroutine has a state of STT, constrained by the following invariants **inv4_1** and **inv4_2**. **inv4_3** and **inv4_4** define the coroutine waiting topic set variable cwt. The topics in the set cwt must be subscribed to by the coroutine.

```
inv4_1: $cst \in crt \rightarrow STT$
inv4_2: $\forall c \cdot c \in crt \wedge cst(c) = active \Rightarrow$
            $(\forall cr \cdot cr \in crt \wedge cr \neq c \wedge$
            $ct(c) = ct(cr) \Rightarrow cst(cr) = sleeping)$
inv4_3: $cwt \in top \leftrightarrow crt$
inv4_4: $cwt \subseteq cstp(era)$
```

We add judged states to the guard conditions of related events to ensure that all coroutines in the same thread have only one coroutine running simultaneously. Two events *wait_msg* and *recv_msg* define the transition of the coroutine's states: the event *wait_msg* changes the running state from active to sleeping, and the event *recv_msg* changes the running state from *sleeping* to *active*.

The active state coroutine requests that the topic of interest's message calls the event *wait_msg*, and the running state changes to *sleeping*. The meaning of *wait_msg* is that an active coroutine initiates a waiting request for certain topics, and the two parameters c and ts represent the coroutine and topic set, respectively. The guard condition $grd1$ requires that the variable c is an active coroutine. The guard condition $grd2$ indicates that ts is a nonempty topic set and that all topics in ts are subscribed to by c. $grd3$ requires that the coroutine c is not the last active coroutine in the system.

```
wait_msg:
  ANY c, ts
  WHERE
    grd1: $c \in crt \wedge$ cst(c) = active
    grd2: $(ts \subseteq top \wedge ts \neq \varnothing) \wedge$
          $(ts \times \{c\} \subseteq cstp(era) \setminus \varnothing)$
    grd3: $(\exists d \cdot d \in crt \wedge d \neq c \wedge$
          $cst(d) \neq sleeping \wedge d \in ran(sc)) \vee$
          $(\exists m, d \cdot d \in crt \wedge d \in ran(sc) \wedge$
          $m \in msg \wedge tms(m) \mapsto$
          $d \in cwt \cup (ts \times \{c\}) \wedge$
          $(m \mapsto d \in (nyd2(onth(tms(m)))) \vee$
          $m \mapsto sc^{-1}(d) \in (th\_rq(ct(d)))))$
  THEN
    act1: $cst(c) := sleeping$
    act2: $cwt := cwt \cup (ts \times \{c\})$
  END
```

The meaning of the receive message event *recv_msg* is that a sleeping coroutine c has received the message m that it is waiting for. If there is no active coroutine in the thread of c at this time, c will receive this message. $grd1$ requires the variable c to be a coroutine. $grd2$ indicates that the message m is sent to the receiving queue of the thread of c. $grd3$ indicates that the topic of message m is subscribed to and awaited by coroutine c. $grd4$ requires that all coroutines in the thread of c are sleeping. The meaning of the next few actions is that the coroutine c receives the message m, then the waiting request of c is removed, and c enters the active state.

```
recv_msg:
  ANY t, m, c
  WHERE
    grd1: $c \in crt$
    grd2: $m \mapsto sc^{-1}(c) \in th\_rq(ct(c))$
    grd3: $(tms(m) \mapsto c \in cstp(era)) \wedge$
          $tms(m) \mapsto c \in cwt)$
    grd4: $\forall cr \cdot cr \in crt \wedge ct(cr) = ct(c)$
          $\Rightarrow cst(cr) = sleeping$
  THEN
    act1: $rms(sc^{-1}(c)) := rms(sc^{-1}(c)) \cup \{m\}$
    act2: $th\_rq(ct(c)) := th\_rq(ct(c)) \setminus$
          $\{m \mapsto sc^{-1}(c)\}$
    act3: $crms(c) := crms(c) \cup \{m\}$
    act4: $cst(c) := active$
    act5: $cwt := cwt \rhd \{c\}$
  END
```

The above method describes the model refinement process that is performed after the abstract model is established. The purpose of model 7 is to support the proof of nature, and the corresponding rewritten requirement is **SAF-2**. Since model 7 does not involve the refinement of any variables or events, it is not described in this section.

VII. VERIFICATION AND VALIDATION

Invariants are proposed to express the system's properties to verify and validate the system and subsequently generate and prove the corresponding proof obligations.

A. VERIFICATION BY PROOF OBLIGATIONS

The interplay between editing models and analyzing their proof obligations are the two components of the Event-B method [8]. Proof obligation ensures that each event preserves the invariants and verifies that the refinement has been performed correctly. The proof obligation represents the backbone of Event-B's ability to demonstrate the correctness regarding some behavioral semantics [34]. The Rodin Platform automatically generates proof obligations, and the rules for generating proof obligations follow the substitutions calculus [7]. Three types of proof obligations are essential in the Event-Bus hybrid model.

1) INVARIANT PRESERVATION PROOF OBLIGATION

The invariant proof obligation ensures that the invariant is always established during the operation of the entire system. Let v be the variables before the event is executed and v' be the variables after the event is executed. Sets s and constant c

constitute the context file. The axioms of s and c are denoted by $P(s, c)$, and the invariant is denoted by $I(s, c, v)$. Let $G(s, c, v)$ be the guard and $R(s, c, v, v')$ be the before-after predicate of a machine's event. The statement to prove to guarantee that the event maintains invariant $I(s, c, v)$ is the following:

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v') \quad (1)$$

2) THEOREM PROOF OBLIGATION

The theorem proof obligation is automatically created to ensure that a stated context and machine theorem are provable from the axioms and invariants. In accordance with which variables are involved in the theory, the theory variant can be divided into $T(s, c, v)$ and $T(s, c)$. When the axioms and invariants are determined, the theorem invariant is satisfied. Its definition is shown below.

$$\begin{aligned} P(s, c) &\Rightarrow T(s, c) \\ P(s, c) \wedge I(s, c, v) &\Rightarrow T(s, c, v) \end{aligned} \quad (2)$$

3) REFINEMENT PROOF OBLIGATION

If machine M refines machine N, M is called the concrete machine and N is the abstract machine. The invariant preservation proof obligation should be established to verify that each concrete event preserves both newly created invariants in the concrete machine and abstract invariants. The concrete event execution does not contradict the corresponding abstract event.

B. CORRECTNESS INVARIANTS

Properties are corresponding proof rules and invariant constraints in Event-B. Two essential properties of concurrent systems are safety and liveness. Safety properties are described as "something bad never happens." Deadlock-free [35] is an essential safety property in the Event-B model. Liveness properties [36] refer to the state of "something good eventually happens." The main property is a description of the liveness of the system.

1) DEADLOCK-FREE

Deadlock in Event-B means there is no way to execute any event when the model is in a given state and the guard conditions of all events are not met. In contrast, a model has at least one event to execute in any state is called deadlock-free. Consider a model with n events, where each event has several guardian conditions. Let $G_i(1 < i < n)$ be the sum of all guardian conditions for the $i - th$ event. $G_i(1 < i < n)$ is true means that event i of the model can execute. The mathematical expression of the model's deadlock-free theory is as follows.

$$P(s, c) \wedge I(s, c, v) \Rightarrow G_1(s, c, v) \vee G_2(s, c, v) \vee \cdots \vee G_n(s, c, v) \quad (3)$$

The implementation model is complicated, and it is difficult to prove the deadlock-free property directly. We solve the problem through a bottom-up proof method. The deadlock-free property is proved at every model level. The proof result of a lower-level model can be utilized as a theorem to assist in proving a higher-level model.

The deadlock-free theorem of the abstract model is denoted as **dlf_0**, which is described as follows. There are nine guard conditions in the abstract model that indicate whether the abstract model is deadlock-free.

dlf_0: $P(s, c) \wedge I(s, c, v) \Rightarrow G_1(s, c, v) \vee G_2(s, c, v) \vee \cdots \vee G_9(s, c, v)$

The deadlock-free theorem of the implementation model is denoted by **dlf**, with the previous-level model's deadlock-free property theorems. The mathematical expression is as follows.

dlf: $dlf\_0, \ldots, dlf\_6 \vdash P(s, c) \wedge I(s, c, v) \Rightarrow G_1(s, c, v) \vee G_2(s, c, v) \vee \cdots \vee G_{16}(s, c, v)$

2) MAIN PROPERTY

The main property is that all messages in the system can be delivered correctly. The main property of the abstract model is described **main_1** as follows. If subscriber s receives message m, s must be interested in the topic when m is sent.

main_1: $\forall s, m \cdot s \in sub \wedge m \in rms(s) \Rightarrow tms(m) \mapsto s \in stp(erm(m))$

All events satisfy the main property before and after running. The main property's description and proof change as the models are refined. We rewrite the main property in the other two refined models.

In model 1, the event *handle_msg* is split into the required event *recv_msg* and the refused event *refu_msg*: the main property must be modified to match the case for both events. The added variable e describes the event to ensure the timeliness of a message. The main property is as follows.

main_2: $\forall s, m \cdot s \in sub \wedge m \in rms(s) \Rightarrow (\exists e \cdot e \in erm(m) \,..\, era \wedge tms(m) \mapsto s \in stp(e))$

Model 6 introduces threads and coroutines to replace publishers and subscribers, respectively. Correspondingly, the invariant of the main property is changed based on the thread and the coroutine. The coroutine c replaces the subscriber s. The main property is as follows.

main_3: $\forall c, m \cdot c \in crt \wedge m \in crms(c) \Rightarrow$
$(\exists e \cdot e \in erm(m) \mathrel{..} era \wedge$
$tms(m) \mapsto c \in cstp(e))$

C. VALIDATION AND STATISTICS

For model validation, the Rodin platform is employed to model and verify Event-Bus. The Rodin platform automatically generates the corresponding proof obligations for each theorem and invariant and can also automatically prove most of the proof obligations, although a few complex proof obligations require manual interactive proof. The proof obligations of the correctness generation invariants are to be proven at each model level.

There are many powerful plugins in Rodin; in this work, we utilize Atelier B, AnimB, and ProB to reduce our workload. Atelier B proves the proof obligations automatically, and ProB is a constraint solver and model checker for the Event-B method. Most of the proof obligations are demonstrated without user intervention. The constraint-solving capabilities of ProB can be applied to model development, deadlock checking, and test-case generation. AnimB animates and simulates models: it allows fully automatic animation and can systematically check a specification for a wide range of errors.

TABLE 3. Proof statistics of the Event-Bus hybrid model.

Model	Number of proof obligations	Automatically discharged	Interactively discharged
M0	40	40(100%)	0(0%)
M1	65	64(98%)	1(2%)
M2	88	74(84%)	14(16%)
M3	101	94(93%)	7(7%)
M4	22	21(95%)	1(5%)
M5	216	195(90%)	21(10%)
M6	78	78(100%)	0(0%)
M7	24	11(46%)	13(54%)
Total	634	577(91%)	57(9%)

Table 3 shows the proof results of the models in the Rodin platform. A total of 634 proof obligations were discharged, of which 577 were discharged automatically, whereas the remaining 57 required manual certification, for an automatic certification rate of 91%. The completion rate of automatic proofs is related to the complexity of the proof obligation. The abstract model is automatically proven because of its simple structure. However, the proof obligations of the deadlock-free property involve all events' guard conditions, and the structure is the most complex. Therefore, we prove the deadlock-free property by hand in every refined model. The powerful plugins of Rodin dramatically help our work by automating most of the tedious mechanical work. Nevertheless, part of the verification work requires human intervention, which can be avoided by more thoughtful modeling and more intelligent automatic proof tools.

VIII. CONCLUSION AND FUTURE WORK

We focus on the modeling and verification of a hybrid model for an IoT OS. Various programming models are analyzed, and the Event-Bus hybrid model is proposed. The Event-B method is applied for modeling and verification of Event-Bus. The refinement strategy partitions all requirements into 8 levels of models. The Event-Bus is verified and validated by proving the corresponding proof obligation in each level of model, and the deadlock-free property and main property are satisfied in each model.

Work on the remaining issues in this paper will be presented in the future. We will focus on automated verification and propose a set of Event-B modeling specifications to further automate the system's proving work. Furthermore, we will explore the automated code generation of Event-B, convert Event-B code into executable C code more efficiently, and guide the code implementation of the Event-Bus IoT OS.

ACKNOWLEDGMENT

The authors would like to thank Abrial (the inventor of the Event-B method) for his help in modeling verification, Z. Zhou for guidance on system design, and the editor and anonymous reviewers for reviewing this article.

REFERENCES

[1] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A survey on resource management in IoT operating systems," *IEEE Access*, vol. 6, pp. 8459–8482, 2018.
[2] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks," in *Proc. Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2008, pp. 233–244.
[3] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2013, pp. 79–80.
[4] N. Wirth and J. Gutknecht, "The oberon system," *Softw., Pract. Exper.*, vol. 19, no. 9, pp. 857–893, 1989.
[5] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An energy-aware resource-centric RTOS for sensor networks," in *Proc. 26th IEEE Int. Real-Time Syst. Symp. (RTSS)*, Dec. 2005, p. 10.
[6] R. Alur, "Formal verification of hybrid systems," in *Proc. 9th ACM Int. Conf. Embedded Softw. (EMSOFT)*, 2011, pp. 273–278.
[7] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2010.
[8] S. Hallerstede, "On the purpose of event-B proof obligations," in *Proc. Int. Conf. Abstract State Mach., B Z*. Berlin, Germany: Springer, 2008, pp. 125–138.
[9] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 6, pp. 447–466, Nov. 2010.
[10] M. M. Gallardo, C. Joubert, P. Merino, and D. Sanán, "A model-extraction approach to verifying concurrent C programs with CADP," *Sci. Comput. Program.*, vol. 77, no. 3, pp. 375–392, Mar. 2012.
[11] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato, "Bounded model checking of multi-threaded C programs via lazy sequentialization," in *Proc. Int. Conf. Comput. Aided Verification*. Cham, Switzerland: Springer, 2014, pp. 585–602.
[12] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2005, pp. 82–97.
[13] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, "SeL4: Formal verification of an OS kernel," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ. (SOSP)*, 2009, pp. 207–220.
[14] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, "Modeling and verifying the code-level OSEK/VDX operating system with CSP," in *Proc. 5th Int. Conf. Theor. Aspects Softw. Eng.*, Aug. 2011, pp. 142–149.

[15] X. Zhu, M. Zhang, J. Guo, X. Li, H. Zhu, and J. He, "Toward a unified executable formal automobile OS kernel and its applications," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 1117–1133, Sep. 2019.

[16] A. Lal and S. Qadeer, "Powering the static driver verifier using corral," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, 2014, pp. 202–212.

[17] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and V. A. Khoroshilov, "Configurable toolset for static verification of operating systems kernel modules," *Program. Comput. Softw.*, vol. 41, no. 1, pp. 49–64, 2015.

[18] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. NSDI*, vol. 4, 2004, p. 12.

[19] A. Liu, A. Alqazzaz, H. Ming, and B. Dharmalingam, "Iotverif: Automatic verification of SSL/TLS certificate for IoT applications," *IEEE Access*, vol. 9, pp. 27038–27050, 2021.

[20] S. Liakh, M. Grace, and X. Jiang, "Analyzing and improving linux kernel memory protection: A model checking approach," in *Proc. 26th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2010, pp. 271–280.

[21] Z. Feng, Z. Yongwang, M. Dianfu, and N. Wensheng, "Fine-grained formal specification and analysis of buddy memory allocation in zephyr RTOS," in *Proc. IEEE 22nd Int. Symp. Real-Time Distrib. Comput. (ISORC)*, May 2019, pp. 10–17.

[22] A. Gawanmeh, "Embedding and verification of ZigBee protocol stack in event-B," *Procedia Comput. Sci.*, vol. 5, pp. 736–741, Jan. 2011.

[23] W. Su, J.-R. Abrial, G. Pu, and B. Fang, "Formal development of a real-time operating system memory manager," in *Proc. 20th Int. Conf. Eng. Complex Comput. Syst. (ICECCS)*, Dec. 2015, pp. 130–139.

[24] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Netw. Appl.*, vol. 10, no. 4, pp. 563–579, Aug. 2005.

[25] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, May 2011.

[26] Z. Vincze, D. Vass, R. Vida, A. Vidács, and A. Telcs, "Adaptive sink mobility in event-driven multi-hop wireless sensor networks," in *Proc. 1st Int. Conf. Integr. Internet Ad Hoc Sensor Netw. (InterSense)*, 2006, p. 13.

[27] L. Saraswat and P. S. Yadav, "A comparative analysis of wireless sensor network operating systems," *Int. J. Eng. Technosci.*, vol. 1, no. 1, pp. 41–47, 2010.

[28] S. M. Trenkwalder, Y. K. Lopes, A. Kolling, A. L. Christensen, R. Prodan, and R. Groß, "OpenSwarm: An event-driven embedded operating system for miniature robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2016, pp. 4483–4490.

[29] W. Dong, C. Chen, X. Liu, Y. Liu, J. Bu, and K. Zheng, "SenSpire OS: A predictable, flexible, and efficient operating system for wireless sensor networks," *IEEE Trans. Comput.*, vol. 60, no. 12, pp. 1788–1801, Dec. 2011.

[30] J. Abrial, *The B-Book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge Univ. Press, 1996.

[31] R.-J. Back, "Refinement calculus, Part II: Parallel and reactive programs," in *Proc. Workshop/School/Symp. REX Project, Res. Educ. Concurrent Syst.* Berlin, Germany: Springer, 1989, pp. 67–93.

[32] J.-R. Abrial and S. Hallerstede, "Refinement, decomposition, and instantiation of discrete models: Application to event-B," *Fundamenta Informaticae*, vol. 77, nos. 1–2, pp. 1–28, 2007.

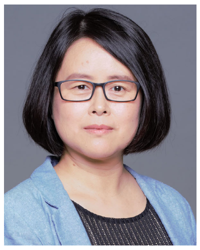
[33] W. Su, J.-R. Abrial, R. Huang, and H. Zhu, "From requirements to development: Methodology and example," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2011, pp. 437–455.

[34] A. S. A. Hadad, C. Ma, and A. A. O. Ahmed, "Formal verification of AADL models by event-B," *IEEE Access*, vol. 8, pp. 72814–72834, 2020.

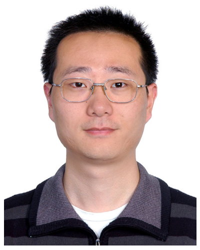
[35] A. Lahouij, L. Hamel, and M. Graiet, "Deadlock-freeness verification of cloud composite services using event-B," in *Proc. OTM Confederated Int. Conf. Move Meaningful Internet Syst.* Cham, Switzerland: Springer, 2018, pp. 604–622.

[36] T. S. Hoang and J.-R. Abrial, "Reasoning about liveness properties in event-B," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2011, pp. 456–471.

YUQIAN GUAN received the bachelor's degree in software engineering from East China Normal University, where he is currently pursuing the master's degree. His research interests include the Internet of Things, formal verification, and operating system memory management.



JIAN GUO is currently an Associate Professor with the Software Engineering Institute, East China Normal University. Her main research interest includes the modeling and analysis of embedded computer systems. In particular, she is interested in the modeling and verification of operating systems. Her methods of interest include model checking, animation, and event-B.



QIN LI is currently an Associate Professor with the Software Engineering Institute, East China Normal University. His research interests include unifying theories of programming, formal modeling and verification of trustworthy software, and trustworthy AI systems and their collaborative behavior.