Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems | School of Computing and Information Systems

12-2023

Refinement-based Specification and Analysis of Multi-core ARINC 653 Using Event-B

Feng ZHANG

Leping ZHANG

Yongwang ZHAO

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the Databases and Information Systems Commons, and the Software Engineering Commons

Citation

ZHANG, Feng; ZHANG, Leping; ZHAO, Yongwang; LIU, Yang; and SUN, Jun. Refinement-based Specification and Analysis of Multi-core ARINC 653 Using Event-B. (2023). *Formal Aspects of Computing*. 35, (4), 1-29.
Available at: https://ink.library.smu.edu.sg/sis_research/8480

This Journal Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Refinement-based Specification and Analysis of Multi-core ARINC 653 Using Event-B

FENG ZHANG, Jiaxing Research Institute, Zhejiang University, China
LEPING ZHANG, School of Computer Science and Engineering, Beihang University, China
YONGWANG ZHAO, School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, China
YANG LIU, School of Computer Science and Engineering, Nanyang Technological University, Singapore
JUN SUN, School of Computing and Information Systems, Singapore Management University, Singapore

ARINC 653 as the de facto standard of partitioning operating systems has been applied in many safety-critical domains. The multi-core version of ARINC 653, ARINC 653 Part 1-4 (Version 4), provides support for services to be utilized with a module that contains multiple processor cores. Formal specification and analysis of this standard document could provide a rigorous specification and uncover concealed errors in the textual description of service requirements. This article proposes a specification method for concurrency on a multi-core platform using Event-B, and a refinement structure for the complicated ARINC 653 Part 1-4 provides a comprehensive, stepwise refinement-based Event-B specification with seven refinement layers and then performs formal proof and analysis in RODIN. We verify that the errors discovered in the single-core version standard (ARINC 653 Part 1-3) also exist in the ARINC 653 Part 1-4 during the formal specification and analysis.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Computer systems organization → Real-time operating systems**; • **Software and its engineering → Software verification and validation;**

Additional Key Words and Phrases: Multi-core ARINC 653, Event-B, refinement, formal specification and analysis

ACM Reference format:
Feng Zhang, Leping Zhang, Yongwang Zhao, Yang Liu, and Jun Sun. 2023. Refinement-based Specification and Analysis of Multi-core ARINC 653 Using Event-B. *Form. Asp. Comput.* 35, 4, Article 24 (November 2023), 29 pages.
https://doi.org/10.1145/3617183

This work has been supported in part by the Natural Science Foundation of China under Grant 62132014 and Zhejiang Science and Technology Plan Project under Grant No. 2022C01045.
Authors' addresses: F. Zhang, Jiaxing Research Institute, Zhejiang University, China; email: zhangfeng@jrizju.com; L. Zhang, School of Computer Science and Engineering, Beihang University, China; email: zhangleping@buaa.edu.cn; Y. Zhao (Corresponding author), School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, China; email: zhaoyw@zju.edu.cn; Y. Liu, School of Computer Science and Engineering, Nanyang Technological University, Singapore; email: yangliu@ntu.edu.sg; J. Sun, School of Computing and Information Systems, Singapore Management University, Singapore; email: junsun@smu.edu.sg.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0934-5043/2023/11-ART24 $15.00
https://doi.org/10.1145/3617183

1 INTRODUCTION

Partitioning Operating Systems (POSs) [33] can provide hard guarantees of temporal and spatial partitioning and isolation. These guarantees can ensure that the effects of any misbehavior are confined to their own components. The SAE-ITC promotes the ARINC 653 standard [5], which standardizes the interface between POSs and application software as well as the system functionality of POSs. ARINC 653 is actually the de-factor standard of POSs. Many ARINC 653-compliant productions, such as the commercial VxWorks653 [37], INTEGRITY-178B [21], LynxOS-178 [38], PikeOS [39], and the open-source production, e.g., POK [13], have been widely applied in many domains from aerospace to automotive.

The ARINC 653 Part 1-4 (Version 4) [5] standardizes APIs for multiprocessor platforms. However, the standard document is described using informal textual language and diagrams, which makes it hard to analyze the consistency and correctness of the standard itself.

A rigorous and machine-checkable specification and analysis of ARINC 653 can help to ensure the correctness of this standard. For concurrency [8] on multi-core platforms, the formal proof of seL4 [9, 16, 25, 40] using Isabelle/HOL [29] is based on the coarse-grained locking [32]. The CertiKOS [18, 19] using Coq [15] builds certified, fine-grained locking concurrent OS kernels that support interleaved execution of kernel/user modules across different layers of abstraction. Nonetheless, CertiKOS doesn't support the POS policy.

There are some works on the single-core ARINC 653 [10, 31, 36, 42]. We have utilized Event-B to construct formal specifications and perform analysis on ARINC 653 [44, 46]. Zhao et al. [46] propose a specification method with seven refinement layers, provide a comprehensive model, and then perform analysis using Rodin [4]. Nevertheless, this article presents the first work that provides a comprehensive, stepwise refinement-based formalization for the multi-core ARINC 653.

The reasons we select Event-B to conduct formalization are as follows: First, the "guard-action" programming of Event-B is appropriate to represent the nested "if-else" frames of ARINC 653. Its structural grammar utilizes many nested "if-else" frames, each of which defines one or several service actions. We can naturally split a service into several atomic events in Event-B. Furthermore, the service actions after an "if-else" judge can be organized as one event, and the "if-else" judgment can be represented as a guard condition. Second, Rodin [4], the IDE of Event-B, supports proof obligations to be automatically generated and provides excellent automatic proof function associated with set theory, which can markedly reduce our proof burden. Lastly, the inductive verification approach avoids the state-space explosion of automatic approaches (e.g., model checking) when verifying complex POSs. Therefore, Event-B has been successfully applied in some industry cases [23, 28].

There are three main challenges in our work. First, a generic method needs to be developed for specifying process concurrencies on a multi-core platform using atomic Event-B events. Second, a specification method needs to be constructed that can model the comprehensive ARINC 653 system, including its various components, communication channels, and a suitable refinement structure for its different modules. Lastly, constructing a comprehensive and complicated model that conforms to the highly complicated ARINC 653 P1-4 standard, which uses over 110 pages of textual descriptions to describe 67 services, will require a significant amount of manpower.

This article presents the following technical contributions:

(1) We propose a generic method to specify process concurrencies on multi-core platforms using atomic Event-B events. We define the core configuration, recognize racy [30] processes that concurrently execute to compete for shared resources, split an ARINC 653 service into several sub-procedures, model each sub-procedure using an Event-B event, and give each event an execution location to denote the execution sequence in a service.

(2) We present a comprehensive, fine-grained, refinement-based Event-B specification for the multi-core ARINC 653 P1-4, which contains seven refinement layers, 300 lines of context code, and more than 8,760 lines of model code in the last-layer machine. This specification covers the system functionality and all the 67 services. As far as we know, this specification is the most comprehensive and complicated formal specification in the literature.[1]
(3) We formally verify a substantial number of safety properties that are specified as 157 invariants and 3,814 proof obligations. We verify that the critical errors disclosed in the single-core ARINC 653 P1-3 (presented in the literature [44]) also exist in ARINC 653 P1-4.

The rest of this article is organized as follows: Section 2 introduces ARINC 653, Event-B, and RODIN. Section 3 proposes a specification method for concurrencies on a multi-core platform for the first challenge, Section 4 proposes the refinement structure and method for the second challenge, Section 5 introduces the Event-B specification of ARINC 653 P1-4 for the third challenge, Section 6 describes project evaluation, proof obligations, and verification results, and Section 7 introduces the current research status. Finally, Section 8 gives the conclusion and the future work. Additionally, we present a notation list and a operator list in Event-B in the appendix.

2 PRELIMINARY

2.1 Arinc 653

ARINC 653 specifies the basic operating environment for application software used within **Integrated Modular Avionics (IMA)**. The multi-core ARINC 653, ARINC 653 Part 1-4 (Version 4), is an extension of the original ARINC 653. It defines a general-purpose APEX (APplication/EXecutive) interface between the **Operating System (OSs)** and the application software. It provides support for the APEX services to be utilized with a core module that contains single or multiple processor cores.

Like the single-core version, the ARINC 653 Part 1–4 also defines a partitioning architecture for safety-critical systems on a computing platform. The major functionalities of POSs are partition and process management, timing management, inter- and intra-partition communication, and health monitoring. It contains the interface requirements between application software and POSs, and services that allow the application software to control the scheduling, communication, and status information of its internal processing elements.

The ARINC 653 Part 1–4 describes the interface requirements using 45 pages of natural language associated with diagrams. And it defines 67 APEX services using 70-page descriptions, which use a specification grammar of a structural natural language. The structural language uses the "if-else" and "error-normal" format, which is illustrated in Table 1.

Each partition in ARINC 653 consists of one or multiple processes that run concurrently and share access to processor cores and other resources. When a partition is assigned multiple cores, multiple processes within this partition can run concurrently on the assigned cores.

ARINC 653 Part 1–4, published in 2015, **supports only multiple processes in one partition that executes concurrently on different processor cores** but can't support that multiple partitions execute on different cores or that multiple processes in different partitions execute on different processor cores concurrently. In one word, this type of concurrency supports that the only various processes in the same partition execute concurrently on different cores, and we call it process-level concurrency.

[1]The Event-B code of the project can be found at https://github.com/zf-zhangfeng/MultiCore653-EventB

Table 1. The ARINC 653 Service SUSPEND_SELF

```
1. procedure SUSPEND_SELF
2.    Input: TIME_OUT; Output: RETURN_CODE
3. error
4.    when (current process owns a mutex or current process is error handler process) ⇒
5.      RETURN_CODE := INVALID_MODE;
6.    when (TIME_OUT is out of range) ⇒ RETURN_CODE := INVALID_PARAM;
7.    when (process is periodic) =⇒ RETURN_CODE := INVALID_MODE;
8. normal
9.    if (TIME_OUT is zero) then RETURN_CODE := NO_ERROR;
10.    else set the current process's state to WAITING;
11.        if (TIME_OUT is finite) then initiate a time counter with duration TIME_OUT;
12.        end if;
13.        ask for process scheduling;
14.       if (expiration of the time-out) then RETURN_CODE := TIMED_OUT;
15.        else RETURN_CODE := NO_ERROR;
16.        end if;
17.   end if;
18. end SUSPEND_SELF;
```

2.2 Event-B & RODIN

Event-B [3] is a formal specification and proof method based on state machines. It utilizes set theory as a modeling notation. Furthermore, it uses the formal refinement to represent systems at different abstraction levels and mathematical proofs to verify consistencies between layers of refinements.

RODIN [4] is the IDE for Event-B, which provides a well-supported environment for system specification, refinements, and proof. It is an industrial-strength tool for creating and analyzing Event-B models. It contains a proof-obligation generator and supports for interactive and automated theorem proving.

The refinement mechanism is the key characteristic of Event-B. Refinement is a verifiable transformation of an abstract specification into a concrete specification and further into implementation. The refinement proof can formally connect the top abstract model with the concrete model, such that the properties of the abstract analysis are formally held by the deployed implementation. Each model in the hierarchy is formally connected to the model above directly via refinement proofs.

Event-B models are described in terms of contexts and machines. Machines contain the **dynamic** parts, which consist of variables, invariants, theorems, variants, and events, whereas contexts contain the **static** parts, which consist of carrier sets, constants, axioms, and theorems (see Section 5.1.1 in [3]). Machines also contain a conjoined list of predicates (invariants) to constrain variables, and state transitions (called Events).

Given a machine M, seeing a context C containing sets s and constants c, an event E of the machine M is represented as Equation (1). E is the event name, x is a list of parameters in the event, v is a list of variables in the machine, $G(s, c, v, x)$ are guards of the event, and v :| BA(s, c, v, x, v') are actions that define how the state variables evolve when the event occurs. Refinement of machines in Event-B provides a means for introducing details about the dynamic

properties.

$$
\begin{aligned}
E \;\widehat{=}\; & \textbf{any } x \\
& \textbf{where } G(s, c, v, x) \\
& \textbf{then } v :| \; BA(s, c, v, x, v')
\end{aligned} \tag{1}
$$

3 A SPECIFICATION METHOD FOR CONCURRENCY ON MULTIPLE CORES

3.1 Basic Elements

Services in ARINC 653 consist of components, attributes, constraints, and actions. These basic elements are represented in Event-B as sets, variables, constants, events, and invariants, respectively. The policy described in this section is similar to that for single-core ARINC 653 [44, 46].

Components are specified as *sets*. The main components in ARINC 653 are processor cores, partitions, processes, communicating components (ports and channels for interpartition communications; buffer, blackboard, semaphore, event, and mutex for intrapartition communications), waiting queues, error handlers, and so on. Partitions, the health monitor (a set of configured tables, e.g., MultiPartitionHMTable, PartitionHMTable), and interpartition communication components (e.g., Port, Channel) are statically configured and initialized during the system/partition initialization phase. Processes and intrapartition communication components (e.g., buffer, semaphore, blackboard, event) are created at system compile time. All these components can be specified using *sets* in the Event-B *context*.

Attributes of components are specified as *axioms*. Attributes are restrictions on components, and they are specified as one or several axioms on their components. For example, partitions and cores in ARINC 653 are specified as sets *PARTITIONS* and *CORES*, respectively, each of which contains one or multiple partitions and cores. Meanwhile, there are some constraints on these two sets. For instance, the set *PARTITIONS* should be finite and contains partitions more than 1 and less than 256. This constraint is specified using two axioms: $finite(PARTITIONS)$, and $card(PARTITIONS) > 0 \land card(PARTITIONS) < 256$.

Relations between components are specified as functions, which may be partial or total functions. For example, a process's current execution state is specified as a partial function: $process\_state \pfun PROCESS\_STATES$, in which the $\pfun$ indicates a partial function, and *PROCESS_STATES* is a set representing process execution states of processes. This partial function indicates that some processes may not have a current execution state because they have not been created yet when the containing partition is in initialization mode.

The relation that each partition has at most one error handler process is specified as a partial function $errorhandler\_ of\_partition \in PARTITIONS \pinj processes$; the $\pinj$ is a partial injection relation.

Actions of components are specified as relations of Event-B sets. In Event-B, a relation is a set of ordered pairs representing a many-to-many mapping between sets. Actions of components describe the behaviors conducted on the sets of elements they operate on, and these actions can be specified as a set of relations.

For example, the action of sending a queuing message *msg* from a queuing port with a message capacity can be specified as a union operation on the set of messages in the port. This can be represented as $msgs\_of\_queuports(port) := msgs\_of\_queuports(port) \cup \{msg \mapsto t\}$. Here, the natural number t is the length of the *msg*, and *msgs_of_queuports* is a function of type: $msgs\_of\_queuingports \in queuing\_ports \to (MESSAGES \pfun \mathbb{N})$, and *queuing_ports* is of type $queuing\_ports \in \mathbb{P}(QueuingPorts)$.

3.2 Core Configurations

The core configuration consists of the core assignment for partitions and the core affinity for processes.

Core assignment for partitions. In the ARINC 653 standard, a partition can be assigned a set of processor cores and is provided exclusive access to the cores assigned to this partition.

One or more processor cores can be assigned to a partition that represents an application and contains one or multiple services. This can be modeled in Event-B as an axiom *axm_assigned_cores_of_part*:

$Cores\_of\_Partition \in PARTITIONS \rightarrow \mathbb{P}_1(cores)$, where $\mathbb{P}_1(cores)$ represents non-empty subsets of *cores*. Essentially, this specification indicates that each partition must own at least one processor core assigned to it. By assigning multiple cores to a partition, processes within the partition can execute concurrently on different cores. This allows for better utilization of resources and improved performance in multi-core systems.

This article complies with the ARINC 653 Part 1–4, which only considers the "*use of multiple processes within a partition scheduled to execute concurrently on different processor cores.*" However, ARINC 653 Part 1–4 does not provide the "*definition of scheduling behaviors associated with multiple partitions scheduled to execute concurrently on different processor cores.*" This means that there is only one currently executing partition at a moment and only processes within this currently executing partition can run concurrently on different processor cores assigned to this partition.

To achieve this characteristic, we introduce the variable currently executing partition represented by $current\_partition \in PARTITIONS$. This variable ensures that only one partition is currently executing at a given time, and only processes within this partition can run concurrently on different processor cores assigned to the partition. In essence, the variable *current_partition* specifies that the current running partition has exclusive access to the cores assigned to it.

Core affinity for processes. A partition can contain multiple processes that run concurrently to provide application functions. When multiple processor cores are assigned to a partition, the partition should determine on which processor core each specific process can run. This is known as the **Process Core Affinity (PCA)** in the standard.

The assignment of cores to processes (PCA) is defined as $processes\_of\_cores \in processes \nrightarrow CORES$, where *processes* and *CORES* are sets of processes and processor cores, respectively. The variable *current_processes_flag* of type CORES × BOOL determines if a core can execute a specific process.

The PCA identifies the processor core a process can run on. The PCA for a specific processor core is used to restrict processes to be eligible for scheduling only on their assigned processor cores. The cores assigned to processes should only be the ones assigned to the partition that the processes belong to. All the processes in a partition should run only on the cores assigned to their partition, which is constrained by an invariant *inv_cores_imply_procandpart* as follows. The $processes\_of\_partition \in processes \nrightarrow PARTITIONS$ represents the partition to which a process belongs.

$\forall proc \cdot$

$proc \in processes \land proc \in dom(processes\_of\_cores) \land proc \in dom(processes\_of\_partition \Rightarrow processes\_of\_cores(proc) \in Cores\_of\_Partition(processes\_of\_partition(proc)))$.

3.3 Execution Locations of Services

In order to specify the current execution phase, i.e., the location, of a service on a given processor core, we introduce the concept of the execution locations for services.

First, we divide a service into several sub-procedures and model each sub-procedure as an event. In the ARINC 653 standard, one service consists of several actions defined textually; each action is described with one clause or adjacent actions with several clauses organized by *if-else* or other structures. These can be specified as one Event-B event, each of which defines one or several actions described by sentences in natural language.

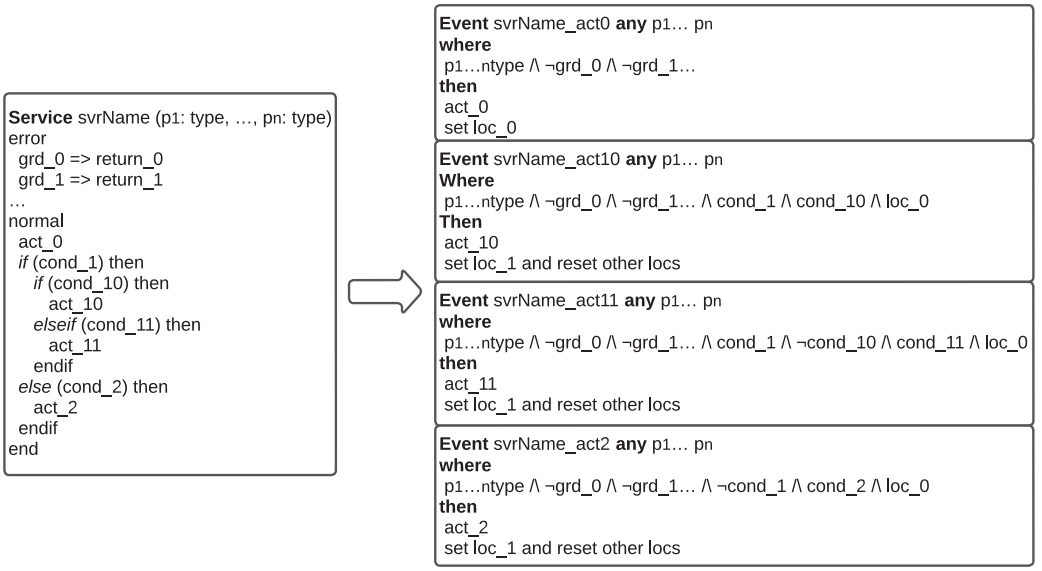

Fig. 1. Specify a service using multiple events.

Second, we identify the execution location of each sub-procedure in a service. We need two parameters: the **execution location** of a sub-procedure on its assigned core and the **execution status** of a core for a service.

This execution location is used to denote which sub-procedure of a service is currently executing and defined in Event-B as $location\_of\_service \in CORES \nrightarrow (Services \times Location)$. The *Location* is a set in Event-B that denotes the current execution location of the service. It is defined as *partition (Location, {loc_i}, {loc_1}, {loc_2}, {loc_3}, . . . ,{loc_r})*, in which *loc_i* represents the initialization and *loc_r* represents the return action (i.e., the last action) of a service. The *loc_1*, *loc_2*, *loc_3*, and so on up to *loc_n* is a sequence of intermediate procedures of service, and the number *n* is determined by the number of sub-procedures.

The execution status is defined as $finished\_core \in CORES \rightarrow BOOL$. It denotes whether a core is running a service. $finished\_core(core) = False$ indicates that the core is currently running a sub-procedure of a service, and it cannot run another service until the return event (identified by *loc_r*) of the current service is completed.

3.4 Criteria of the Specification Method

A general structure of a service in the APEX grammar is illustrated in the left part of Figure 1. The *error* part describes error handling due to incorrect values of actual input parameters, which are robust handling. ARINC 653 uses textual language associated with diagrams to explain the service actions. These textual descriptions use compound statements "if-else" and "SEQUENCE" to describe complex structures, and simple actions are described in natural language.

Then, we propose a criterion of the specification method that derives from our previous work for the single-core ARINC 653 [42, 44, 46].

First, an Event-B event is used to describe a single action in the ARINC 653 service. A description in an ARINC 653 service can serve as a single action. For example, in Table 2, the description "*set the process state to DORMANT*" can be modeled by the Event-B event "*create_process_dormant,*" illustrated in Table 4. Similarly, each of the other sentences in the ARINC 653 service can also be modeled by an event.

Second, the "guard-action" structure of Event-B is used to represent the nested "if-else" frames of the ARINC 653 services. The structural grammar of the ARINC 653 uses many nested "if-else" frames, each of which defines one or several service actions. For example, in Figure 1, only if the error conditions are negative and the judgments ¬ *cond_1* and *cond_10* hold will the action *act_10* be triggered. The triggering conditions of an event correspond to the "if-else" frames of the ARINC 653 services.

Third, a service is split into several atomic Event-B events with the above two policies. The service action after an "if-else" judgment can be organized as one event, and the "if-else" judgment can be represented as a guard condition.

Lastly, an APEX service with multiple actions is decomposed into a set of events with non-intersect guards.

For the essence of the "if-else" semantics, one "if-else" clause uses one location because only one guard will hold and only one action will be enabled. This policy is present in the example in Figure 1; we can see that the actions *act_10*, *act_11*, and *act_2* share one execution location *loc_0*. Only one action (e.g., *act_10*, *act_11*, or *act_2*) can execute when its corresponding guards hold. So we can use an event to represent an action in an "if-else" clause. The parameters as service inputs are packed into the *any* part. Conditions (*grd_0*, *grd_1*, etc.) in the error part indicate incorrect inputs of parameters, and the negations these conditions are represented as guards in Event-B.

3.5 Fine-grained Concurrency

The specification method in our work demonstrates the fine-grained concurrency.

First, services of a partition can execute **in parallel** with each other across all processor cores if they do not compete for shared resources. This is made possible through the core assignment mechanism for partitions and the core affinity mechanism for processes, as described in Section 3.2. The core configuration allows a partition to be assigned a set of cores. Although there is only one currently executing partition at a moment, different processes within a partition can run in parallel on their respective assigned cores.

Second, racy [30] services attempting to compete for shared resources execute **concurrently**. Processes are considered racy if they attempt to access the same shared resources at the same time. Otherwise they are race-free. When a running service attempts to access an unavailable shared resource, this service will be suspended and the OS will invoke the scheduler to allow another eligible service to preempt execution. Once the shared resource becomes available, the suspended service can resume its execution.

For example, let us consider the service SEND_BUFFER. If the message buffer is full and the wait-time argument is not zero, a process (named *P0*) that calls this service will be blocked. The OS will insert this process in the buffer's process queue. Once another process receives and removes some messages from this buffer, thereby freeing up space, the blocked process *P0* will be resumed and execute the remaining actions. This mode is described in detail in Section 4.4. Another example is the service SUSPEND_SELF outlined in Table 1. In this case, the current process will be suspended until either the timeout value expires or another process resumes this suspended process. Subsequently, the OS asks for rescheduling to allow another eligible process to preempt execution.

Based on the execution semantics, services such as SEND_BUFFER, SUSPEND_SELF, and similar ones should be split into at least two parts at the point of scheduling: the first part implements the actions before scheduling, including the scheduling itself, while the second part implements the actions after scheduling, excluding the scheduling.

Lastly, in fact, we employ a more **fine-grained** approach to split and model all ARINC 653 services.

Table 2. The Service CREATE_PROCESS

procedure CREATE_PROCESS
(in: *ATTRIBUTES*; out: *PROCESS_ID*, *RETURN_CODE*)
error
...
normal
create a new process with the process attributes - *ATTRIBUTES*;
set the process state to *DORMANT*;
initialize process context, unique process index, and stack;
set the process's core affinity to the core affinity value;
PROCESS_ID := unique ID assigned to the created process;
RETURN_CODE := *NO_ERROR*;

Table 3. The Event create_process_init

create_process_init $\widehat{=}$ **any** *part proc core service*
where
...
@grd006: finished_core(*core*) = TRUE
@grd007: *service* = Create_Process
then
@act001: finished_core(*core*) := FALSE
@act002: location_of_service(*core*) := *service* ↦ loc_i
@act003: processes := processes ∪ {*proc*}
@act004: processes_of_partition(*proc*) := part
...

Table 4. The Event create_process_dormant

create_process_dormant $\widehat{=}$ **any** *part proc core*
where
...
@grd004: finished_core(core) = False
@grd005: location_of_service(*core*) = Create_Process ↦ loc_i
...
then
@act001: location_of_service(*core*) := Create_Process ↦ loc_1
@act002: process_state(*proc*) := PS_Dormant
...

Table 5. The Event create_process_return

create_process_return $\widehat{=}$ **any** *part proc core*
where
...
@grd004: finished_core(core) = False
@grd005: location_of_service(*core*) = Create_Process ↦ loc_2
...
then
@act001: location_of_service(*core*) := Create_Process ↦ loc_r
@act002: finished_core(core) = True
...

We utilize atomic Event-B events to describe individual statements or actions within an ARINC 653 service, as explained in Sections 3.3 and 3.4. For example, the SUSPEND_SELF shown in Table 1 is modeled by 5 combined Event-B events, while the SEND_BUFFER utilizes 10 combined Event-B events as illustrated in Figure 8 and described in Section 4.4.

3.6 An Example

We illustrate an example using the service CREATE_PROCESS in Table 2. In this example, we focus on the concurrency-related elements only and ignore the functionality of a service.

(1) Construct an initialization event for the first clause to identify the entry of a service with the suffix *_init* and located at *loc_i*. For example, the initialization event *create_process_init*, illustrated in Table 3, serves two main purposes:

First, it checks whether the core can execute this service by verifying if *finished_core(core) = True* holds (3.grd006). If not, it means the core is currently running another service and cannot run this one.

Second, it sets the status of the core to indicate that this core is running this service. This consists of two actions. It first sets *finished_core(*core*)* (Table 3.act001) to False to denote that the core is running a service and can't run other services. Then, it sets the execution location of the current service to *loc_i* (Table 3.act002). The other actions (act003 and act004 in Table 3) are related to the functionality of the service CREATE_PROCESS.

(2) To model the intermediate actions of a service, we use an event and identify its execution location with a sequence number such as *loc_1*, *loc_2*, *loc_3*, and so on. For example, the intermediate functional event *create_process_dormant* in Table 4 models the second-line description "set the process state to DORMANT" in the example service in Table 2.

To handle these intermediate actions, we need to address two issues:

First, verify whether the core can execute this intermediate service by checking the *finished_core(core)* and *location_of_service*. Unlike an initialization event with *finished_core(core) = True*, on the contrary, the *finished_core(core)* for an intermediate event should be False (Table 4.grd004). Additionally, the execution location, *location_of_service*, should indicate whether the core is running the previous event. For example, if the candidate event is the first intermediate

Algorithm 1 The APEX2EvB Translation Algorithm

Input: *evts, stmt*: events and statements in the ARINC 653;
spec: <ζ, P, E, S>.
Output: Machines and Contexts in Event-B.

```
1: function TRANSLATE_STMT((evts, stmt))[42]
2:     switch stmt do
3:         case ACT act
4:             add the action act to end of action list of each event in evts;
5:             return evts;
6:         case st1;st2
7:             evts' ← translate(evts, st1);
8:             return translate(evts', st2);
9:         case IF cond THEN st1 ELSE st2
10:            evts' ← duplicate of (evts);
11:            add the "cond" to end of guard list of each event in evts;
12:            evts ← translate(evts, st1);
13:            add the "¬cond" to end of guard list of each event in evts';
14:            evts' ← translate(evts', st2);
15:            return evts ∪ evts';
16:        case IF cond THEN st
17:            evts' ← duplicate of (evts);
18:            add the "cond" to end of guard list of each event in evts;
19:            evts ← translate(evts, st1);
20:            add the "¬cond" to end of guard list of each event in evts';
21:            return evts ∪ evts';
22: end function
23: function CHECK_CORE_AFFINITY_STAUS(proc, part, core, evt)
24:        /*check for the proc's Core Affinity:*/
25:        Core_Affinity(proc) = core;
26:        core ∈ Cores_Assignment(part);
27:        /*check for the core's execution status:*/
28:        IF the evt is a beginning procedure of its service;
29:        /*the evt is an intermediate procedure of its service;*/
30:        THEN Core_Free(core) = True;
31:        /*the evt is an intermediate procedure of its service;*/
32:        ELSE Core_Free(core) = False;
33: end function
34: function SET_CORE_STAUS(core, evt)
35:        /*the evt is a last procedure of its service*/
36:        IF;
37:        THEN Core_Free(core) := True;
38:        /*the evt is an init or an intermediate procedure of its service;*/
39:        ELSE Core_Free(core) := False;
40: end function
41: function TRANSLATE_SERVICE(spec)
42:     evts ← <ζ, ϕ, ϕ>;
43:     evts ← translate(evts, S);
44:     add ∧_i (¬E_i) to guard list of each event in evts;
45:     add CHECK_CORE_AFFINITY_Staus to guard list of each event in evts;
46:     add SET_CORE_Staus to action list of each event in evts;
47:     return {ev. ev ∈ evts ∧ ev.α ≠ ϕ};
48: end function
```

Fig. 2. The APEX2EvB translation algorithm.

event, the current location should be *loc_i* (Table 4.grd005), and if the candidate event is the second intermediate event, the current location should be *loc_1*.

Second, set the status of the core to indicate that it is running an intermediate event. In this example, we should set the execution location to *loc_1* (Table 4.act001) to indicate that the core is running the first intermediate event of the service.

(3) Model the return action of a service by an event whose name is suffixed with *_return* and whose location is identified as *loc_r*. For example, the return event *create_process_return* illustrated in Table 5 corresponds to the fifth and sixth lines in the example service CREATE_PROCESS.

The return action needs to tackle two affairs: (1) it should check whether the core can run this return action, i.e., check that whether the *finished_core(core)* is false (grd004) and whether *location_of_service* denotes the last intermediate event (in this example, the last intermediate event's location is *loc_2* in grd005), and (2) it needs to set the core's status to a value denoting the completion of all actions of the current service and allows it to run other services. This needs to set the execution location to *loc_r* (Table 5.act001).

3.7 Translating Methods from ARINC 653 to Event-B

We designed an algorithm to guide translations from APEX services to Event-B models as shown in Figure 2. We first give a simple abstract syntax for APEX service requirements in the above expression, where "**ACT** *act*" is a simple action, and "*c*; *c*" is a sequential composition statement:

$$
\begin{aligned}
c \ ::= \ & \textbf{ACT}\ act \mid \\
& c; c \mid \\
& \textbf{IF}\ cond\ \textbf{\textit{THEN}}\ c \mid \\
& \textbf{IF}\ cond\ \textbf{THEN}\ c\ \textbf{ELSE}\ c.
\end{aligned}
\tag{2}
$$

In this algorithm, the function TRANSLATE_STMT is from our prior work [44, 46].

The function CHECK_CORE_AFFINITY_STATUS is to check out the core affinity for processes and the running status of cores.

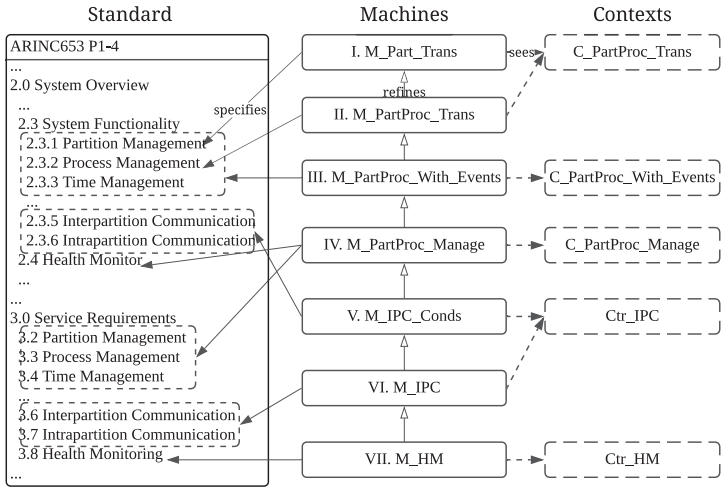

Fig. 3. Structure of the multi-core ARINC 653 along with its Event-B specification [44, 46].

The function SET_CORE_STATUS is to set the core's status at a corresponding value after executing a service.

An event is a tuple $\langle \iota, p, \sigma, \alpha \rangle$, in which ι is the name of the event, p is a list of parameters, σ is a list of guards, and α is a list of actions. We duplicate each event into $evts'$ for each "IF" statement and translate the "IF" conditions as guards.

The function TRANSLATE_SERVICE translates a service requirement *spec* into a final set of events in Event-B by invoking the above sub-functions. A service requirement can be abstracted into a tuple *spec*: <ζ, P, E, S>, where ζ is the service name, P is the input parameter list of this service, E represents error conditions in the error part, and S is a statement of the service's normal action part. The TRANSLATE_SERVICE assigns the parameters of a service to the parameter list of each event and adds the negation of the conditions of the error part to the guard list in the Event-B machine.

4 A REFINEMENT STRUCTURE

The ARINC 653 Part 1–4 standard uses more than 110 pages of textual descriptions to describe its 67 services; it should be formalized with refinement-based methods.

4.1 Refinement Structure for the Entire System

We first introduce the refinement structure in our work, illustrated in Figure 3 (which derives from [44, 46]). This refinement structure is built according to the structure of the standard document.

In the standard, Section 1 and Sections 2.1, 2.2, and 3.1 provide an overview of ARINC 653 from different perspectives. Sections 2.3 and 2.4 describe the functionality of the ARINC 653 and its interface toward application software. Section 3 specifies service requests corresponding to the functionality. Sections 2.5 and 5 define the information and data format of the system configuration for integration and deployment.

The first machine `M_Part_Trans` specifies the partition operating modes, namely *COLD_START*, *WARM_START*, *NORMAL*, and *IDLE*. The abstract mode transitions are demonstrated in Section 2.3.1.4 and Figure 2.3.1.4 in the standard.

The second machine `M_PartProc_Trans` models process states, namely *Dormant*, *Ready*, *Waiting*, *Suspend*, *WaitandSuspend*, *Running*, and *Faulted*. Note that the states *Waiting*, *Suspend*, and

WaitandSuspend are refined from the state *Waiting* in the standard. The reasonability of the introduction of these three states is described in Section 5.2.1. Moreover, this layer specifies state transitions under certain partition modes as described in Section 2.3.1.4.2.2 in the standard. For example, a process at state *Ready* in a partition at *Normal* mode can progress to *Dormant* or *Suspend*.

The third machine `M_PartProc_withEvents` introduces process periods and specifies period-related process actions. Meanwhile, this machine specifies *start*, *stop*, *resume*, and so forth, multiple services for processes. This layer mainly describes (i.e., refines) the abstract state transitions defined in the second layer. For example, the service *stop_self* in the third layer models only that this service will turn the current *running* process in a *normal* partition into *dormant*.

The fourth machine `M_PartProc_Manage` introduces process priorities along with the specific service (*set_priority*), a concrete two-level scheduling policy for partitions and processes, timing management services, and so on.

The fifth machine `M_IPC_Conds` introduces *sampling* and *queueing* modes for interpartition communications. Meanwhile, this layer introduces the *blackboard* and *buffer* for intrapartition communications.

The sixth machine `M_IPC` supplements *semaphore*, *event*, and `Mutex` for control-flow communications, which are used for process synchronization or process concurrencies. Furthermore, it introduces the refresh periodic for sampling ports, queuing discipline for interpartition communication ports and events and buffers.

The last machine `M_HM` introduces health monitoring services. Then, the refinement structure is completed.

4.2 Refinement Structure for Partition Managements

Partition management services affect not only partition modes but also processes, inter- and intrapartition communications, and other resources in the operated partition. Therefore, we regard the partition management as the core module and construct the partition module first.

The refinement of the partition module through the top three layers is illustrated in Figure 4. Once the third layer has been reached, we do not introduce new events for partition management services. We only refine existing partition elements gradually, such as partition time windows, partition core assignments, and processes in partitions.

The first layer uses an event *partition_mode_transition* to specify abstract partition mode transitions described on page 10 of ARINC 653 P1–4. In the second layer, nine events are introduced to describe concrete partition mode transitions with partition core assignments and process core affinities. The third layer introduces events to model the service *SET_PARTITION_MODE* with periodic and aperiodic processes. Subsequent layers refine the partition service gradually by adding processes, inter- and intrapartition communications, health monitoring, and other factors.

4.3 Refinement Structure for Process Managements

The second layer introduces the process. This layer consists of events that are used to model process scheduling and process state transitions. This layer also models the service SET_PRIORITY using the specification method for process concurrencies described in Section 3.3. The third layer models some other process management services, such as STOP, SUSPEND, RESUME, and so forth. All these services also refine the process-state-transition event. The third layer has not yet introduced the process priority, and we will introduce the service SET_PRIORITY in the fourth layer.

For example, the service SUSPEND, illustrated in Figure 5, is refined by four events. These four events model the entry of the service, setting the process to DORMANT, rescheduling the core, and returning the service, respectively. Another service START in Figure 5 is refined by 16 events.

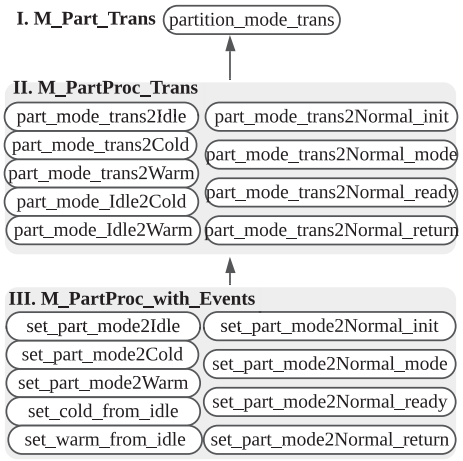

Fig. 4. Refinement of partition-mode transitions.

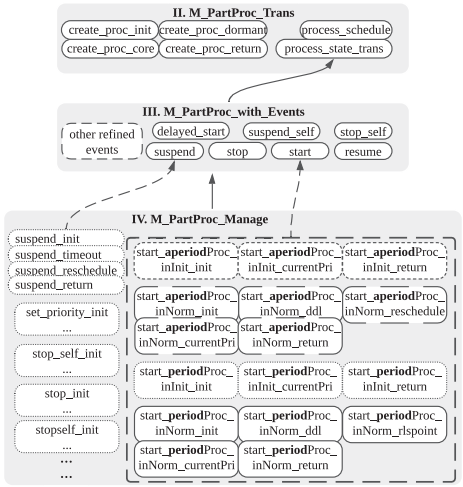

Fig. 5. Refinement of process-state transitions.

This service takes into consideration the periodic and aperiodic processes started in a NORMAL partition or an initialization mode partition, set priority and release points, and so forth. In the subsequent layers, each layer will refine the process management services and introduce new elements for processes.

4.4 Refinement Structure for Inter- and Intrapartition Communications

Regardless of the type of communication (namely queueing or sampling ports, as well as buffers, blackboards, semaphores, mutexes, or events), there is a general characteristic for communication behaviors, the requesting-waiting-scheduling procedures for currently unavailable resources.

The general procedures for communications are based on the following scenarios:

(1) The requested resource is available and no processes are waiting for this resource. When a process issues a request in this situation, such as the service SEND_BUFFER, the service will perform the sending action immediately and the message will be sent out successfully.
(2) The requested resource is available but there exist processes waiting for this resource.
(3) The current resource is unavailable. In this situation, the application will wait for this resource and request to reschedule.

Based on the communication characteristics, we can abstract communication actions as three types of events in the third layer:

(1) The requested resource is currently available and then performs communication actions immediately, such as sending or receiving actions.
(2) The requested resource is current busy, namely the event *req_busy_resource* in Figure 6.
(3) The requested resource becomes available after rescheduling (the event *resource_become_available* in Figure 7). In Figures 6 and 7, we list concrete event refinement of the abstract resource-requesting events *req_busy_resource* and *resource_become_available*.

The fifth layer refines each abstract event into concrete communication events, namely the queuing and sampling, blackboards, buffers, and mutexes.

We use the service SEND_BUFFER in Figure 8 as an example to describe the refinement process.

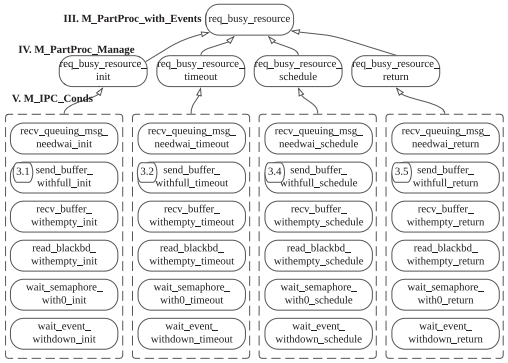

Fig. 6. Refinement of events for requiring an unavailable resource.

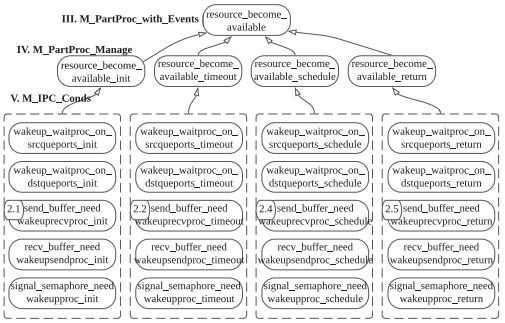

Fig. 7. Refinement of events for waking up one process.

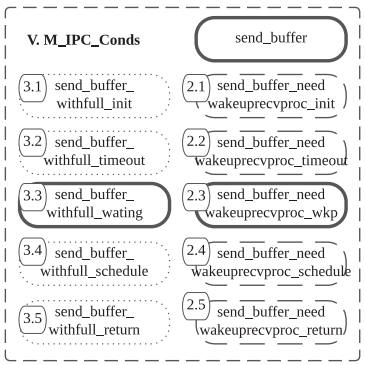

Fig. 8. The Event composition for the service SEND_BUFFER.

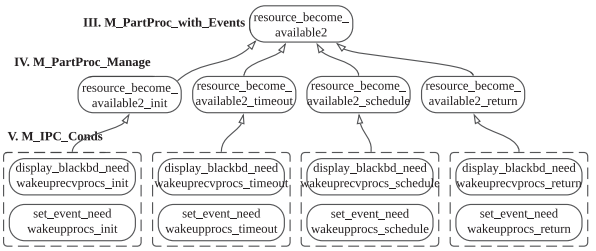

Fig. 9. Refinement of events for waking up multiple processes.

(1) In the first situation, where the requested resource is available and no processes are waiting for this resource, the service will perform the sending event *send_buffer* immediately.

(2) In the second situation, the requested action where the event is *req_busy_resource* will be refined by five sub-procedures:

- The event *2.1* removes the first waiting process from the resource's process queue.
- The event *2.2* handles the timeout value of the waiting process.
- The event *2.3* wakes up this process, removes this process from the WAITING list, and sets the process at READY, unless another process has suspended it.
- The event *2.4* performs rescheduling.
- The event *2.5* executes the remaining actions.

(3) In the third situation, the requested event *req_become_available* will be divided into five sub-procedures: The event *3.1* initializes the context. The event *3.2* sets the timeout type, namely *timed-wait*, *wait forever*, or *no-wait*. The event *3.3* sets the requesting process at WAITING and then inserts this requesting process into the requested port's waiting list. The event *3.4* performs rescheduling. The event *3.5* resumes the suspended process and continues to perform its remaining actions. The refinement relations are illustrated in Figure 8.

We can see from Figures 6 and 7 that these refinement events lack two sets of events: the waking-up action of waiting processes that are blocked on blackboards or on events. The reason we don't

Table 6. Abstract Partition-transition Paths

partition_mode_transition $\widehat{=}$ **any** *part newm* **where**
@grd001: *part* ∈ PARTITIONS
@grd002: *newm* ∈ PARTITION_MODES
@grd003: partition_mode(*part*)=PM_IDLE ⇒ *newm*=PM_WAMR_START ∨ *newm*=PM_COLD_START
@grd004: partition_mode(*part*)=PM_COLD_START ⇒ *newm*=PM_IDLE ∨ *newm*=PM_COLD_START ∨ emphnewm=PM_NORMAL
@grd005: partition_mode(*part*)=PM_WAMR_START ⇒ *newm*=PM_IDLE ∨ *newm*=PM_COLD_START ∨ *newm*=PM_WAMR_START ∨ *newm*=PM_NORMAL
@grd006: partition_mode(*part*)=PM_WAMR_START ⇒ *newm*=PM_IDLE ∨ *newm*=PM_COLD_START ∨ *newm*=PM_WAMR_START
then
@act001: partition_mode(*part*) := *newm*

classify these two types of events into the above refinement structure in Figure 7 is that we should wake up all the processes waiting for blackboards or events, whereas we only wake up the first waiting process blocked on queueing ports, mutexes, or semaphores. Therefore, another abstract event *resource_become_available2* was designed to wake up all of the waiting processes. This abstract event is different from the abstract one *resource_become_available* that wakes up only the first waiting processes, which is illustrated in Figure 9.

Then, we construct the refinement structure for all the inter- and intrapartition communication services.

5 A COMPREHENSIVE EVENT-B SPECIFICATION

This section introduces the comprehensive Event-B specification.

5.1 Partition Managements

A partition is a program unit of an application designed to satisfy these partitioning constraints.

5.1.1 Partition Mode Transitions. We first specify the abstract partition-mode transition paths in the top machine `M_Part_Trans`, which is defined in the event *partition_mode_transition* in Table 6.

We introduce process state transitions under certain partition modes in the second machine. However, these events about partitions are still abstract. In the third machine, we start to specify concrete partition management services associated with periodic/aperiodic processes.

The partition's operating mode is the current execution state of a partition. There exist four types of partition modes: *IDLE*, *NORMAL*, *COLD_START*, and *WARM_START*. The partition mode transition paths specified in Table 6 correspond to Figure 2.3.1.4 in the ARINC 653 standard.

5.1.2 Partition Scheduling. The scheduling of partitions in a module is strictly static and deterministic over time, which is a round-robin scheduling policy. The module generates the time-based scheduling based on the predefined module configuration. Each partition is then scheduled according to its respective partition time windows.

The static scheduling configuration is defined in the context *C_Part_Proc_Manage*, as listed in Table 7. In this context, the constant *partition2num* in the axiom *axm_partitionID:* $partition2num \in PARTITIONS \rightarrowtail \mathbb{N}$ first assigns an ID to each partition using a total bijection. Then, the constant *part_sched_list* in the axiom *axm_part_sched_list:* $part\_sched\_list \in \mathbb{N} \rightarrow (\mathbb{N} \times \mathbb{N})$ defines the static scheduling list. In the axiom *axm_part_sched_list*, the first parameter with the nature type is the partition ID; the second and the third one are the offsets and durations of a partition, respectively. Axioms *axm_part_sched_list1* and *axm_part_sched_list2* present the constraints on the partition configuration; the former shows that each partition has an offset and a time capacity (duration), while the latter shows that each partition's offset is more than or equal

Table 7. The Static Configuration of Partition Scheduling

C_Part_Proc_Manage
axioms
@axm_partitionID: partition2num ∈ PARTITIONS ⤖ ℕ
@axm_part_sched_list: part_sched_list ∈ ℕ → (ℕ × ℕ)
@axm_part_sched_list1: ∀p · p<card(PARTITIONS) ⇒ (∃offset,dur · part_sched_list(p) = (offset ↦ dur))
@axm_part_sched_list2: ∀p · p<card(PARTITIONS) ∧ (p+1)<card(PARTITIONS) ⇒
(∃offset,dur,offset_1,dur_1 · part_sched_list(p) = (offset↦ dur) ∧ part_sched_list(p+1) = (offset_1↦ dur_1) ∧
offset_1≥ offset+dur)
@axm_major_time_window_value:
∃offset,dur · part_sched_list(card(PARTITIONS)-1) = (offset↦ dur) ⇒ offset+dur=majorFrame
@axm_periodicStartPoint: periodicStartPoint ∈ ℕ → ℕ
@axm_periodicStartPoint1: ∀p · p<card(PARTITIONS) ⇒ (∃offset, dur, periodic_start_point ·
part_sched_list(p)=(offset↦ dur) ∧ periodicStartPoint(p)=periodic_start_point ∧ periodic_start_point≥ offset ∧
periodic_start_point < offset+dur)

Table 8. The Partition Scheduling Event

partition_schedule any *part*
where
@grd001: *part* ∈ PARTITIONS
@grd002: partition_mode(*part*) = PM_NORMAL ∨ partition_mode(*part*) = PM_COLD_START ∨
partition_mode(*part*) = PM_WARM_START
@grd101: need_reschedule = TRUE
@grd102: ∃offset, dur· part_sched_list(partition2num(part)) =
(offset↦ dur) ∧ clock_tick mod majorFrame ≥ offset ∧ clock_tick mod majorFrame < offset+dur
then
@act101: need_reschedule := FALSE
@act102: current_partition := part
@act103: need_procresch := need_procresch ⩤ (Cores_of_Partition(part) × {TRUE})

to the prior partition's offset plus its duration. Axiom *axm_major_time_window_value* presents the value of a module's major frame time, which is the last partition's offset plus its duration in the configuration list. Axioms *axm_periodicStartPoint* and *axm_periodicStartPoint1* present the release point of periodic processes.

The action of the partition scheduling is specified by an event *partition_schedule*, as listed in Table 8. In Table 8, the *grd002* represents that partitions in NORMAL mode and initialization mode (COLD_START or WARM_START) can be scheduled. The *Grd101* is a flag for partition scheduling. The *grd102* is the core algorithm that determines the current partition ID based on the current clock time, and the *act102* sets this partition ID as the current partition. The algorithm in the *Grd102* guarantees that the current clock time is within the voted partition's time window; i.e., the value of the current clock time should be between the value of the current partition's offset and the value of its offset plus its duration. The system should perform process scheduling following the partition scheduling. The action *act103* sets the process rescheduling flag to TRUE for the cores assigned to the new partition ID. Then the system will perform process scheduling in the event *partition_schedule()*.

5.1.3 Partition Services. In Figure 4, we can see that the partition service SET_PARTITION_MODE is specified as nine refined events. We present only the event *set_part_mode2Normal_ready* as illustrated in Table 9. This event turns the partition to NORMAL mode, sets all previously started aperiodic processes to READY, and sets all previously delay-started aperiodic processes to WAITING. In this event, *stperprocs* represents **st**arted **per**iodic **proc**esse**s**, *dstperprocs* represents **d**elayed **st**arted **per**iodic **proc**esse**s**, *staperprocs* represents **st**arted **ape**riodic **proc**esse**s**, and *dstaperprocs* represents **d**elayed **st**arted **aper**iodic **proc**esse**s**.

Table 9. The Partition Event *set_part_mode2Normal_ready*

set_part_mode2Normal_ready $\widehat{=}$
any *part procs procs2 procsstate core nrlt stperprocs dstperprocs staperprocs dstaperprocs*
where
...
@grd003: *procs* = processes_of_partition^{-1}[{*part*}] ∩ process_state^{-1}[{PS_Waiting}]
@grd005: *procsstate* ∈ *procs* → {PS_Waiting, PS_Ready}
@grd006: *core* ∈ CORES ∩ dom(location_of_service)
@grd007: location_of_service(*core*) = Set_Normal ↦ loc_1
@grd009: ¬(location_of_service(*core*) = Set_Normal ↦ loc_1 ∧ finished_*core*(*core*) = FALSE)
...
@grd203:
stperprocs = (*procs* \ period_of_process^{-1}[{INFINITE_TIME_VALUE}])∩ process_wait_type^{-1}[{PROC_WAIT_PARTITIONNORMAL}]
@grd204:
dstperprocs = (*procs* \ period_of_process^{-1}[{INFINITE_TIME_VALUE}])∩ process_wait_type^{-1}[{PROC_WAIT_DELAY}]
@grd205:
staperprocs = *procs* ∩ period_of_process^{-1}[{INFINITE_TIME_VALUE}] ∩ process_wait_type^{-1}[{PROC_WAIT_PARTITIONNORMAL}]
@grd206:
dstaperprocs = *procs* ∩ period_of_process^{-1}[{INFINITE_TIME_VALUE}] ∩ process_wait_type^{-1}[{PROC_WAIT_DELAY}]
@grd207: *nrlt* ∈ *stperprocs* → ℕ
@grd208: ∀p,x,y,b · (p∈ *stperprocs* ∧ ((x↦ y)↦ b) firstperiodicprocstart_timeWindow_of_Partition(*part*) ⇒
nrlt(p) = ((clock_tick*ONE_TICK_TIME)/majorFrame+1)*majorFrame + x)
@grd209: *procsstate* = (*staperprocs* ×{PS_Ready}) ∪ ((*dstaperprocs* ∪ *stperprocs* ∪ *dstperprocs*)×{PS_Waiting})
@grd211: ¬(location_of_service2(*core*) = Set_Normal ↦ loc_1 ∧ finished_*core*(*core*) = FALSE)
then
@act001: location_of_service(*core*) := Set_Normal ↦ loc_2
@act002: process_state := (process_state ⩤ *procsstate*) ⩤ (*procs2* × {PS_Suspend})
@act201: location_of_service2(*core*) := Set_Normal ↦ loc_2
@act202: setnorm_wait_*procs*(*core*) := *procs*
@act203: setnorm_susp_*procs*(*core*) := *procs2*
@act204: releasepoint_of_process := releasepoint_of_process ⩤ *nrlt*

Table 10. Abstract Process-transition Paths under Certain Partition Modes

process_state_transition $\widehat{=}$ **any** *part proc newstate core* **where**
...
@grd005: processes_of_partition(*proc*) = *part*
@grd006: partition_mode(*part*) =
PM_COLD_START ∨ partition_mode(*part*) = PM_WARM_START ∨ partition_mode(*part*) = PM_NORMAL
@grd007: ((partition_mode(*part*)=PM_COLD_START ∨ partition_mode(*part*)=PM_WARM_START) ∧
process_state(*proc*)=PS_Dormant) ⇒ *newstate*=PS_Waiting
@grd008: ((partition_mode(*part*)=PM_COLD_START ∨ partition_mode(*part*)=PM_WARM_START) ∧
process_state(proc)=PS_Waiting) ⇒ (*newstate*=PS_Dormant ∨ *newstate*=PS_WaitandSuspend)
@grd009: ((partition_mode(*part*)=PM_COLD_START ∨ partition_mode(*part*)=PM_WARM_START) ∧
process_state(proc)=PS_WaitandSuspendg) ⇒ (*newstate*=PS_Waiting ∨ *newstate*=PS_Dormant)
...
@grd0016: (partition_mode(part)=PM_NORMAL ∧ process_state(*proc*)=PS_Faulted) ⇒ *newstate* = PS_Dormant
then
@act001: process_state(*proc*) := *newstate*

5.2 Process Managements

5.2.1 Process State Transitions. A partition consists of one or multiple processes that combine dynamically and execute concurrently to provide functions associated with that partition's resources.

We first introduce the concept of processes in the second-layer machine `M_PartProc_Trans`. This layer introduces process states, the allocation of processes to partitions, and the core configuration for partitions and processes. Additionally, we specify the service CREATE_PROCESS and define the key event *process_state_transition*, as illustrated in Table 10. The key event specifies process state transition paths under certain partition modes.

The process state *Waiting* in ARINC 653 implies three situations. To distinguish between these three situations, we introduce three state modes to represent *Waiting*: *Suspended*, *Waiting*, and *WaitandSuspend*. These state modes are described as follows:

Table 11. The Process-scheduling Event

process_schedule $\widehat{=}$ **refines** process_schedule **any** *part proc core errproc* where
...
@grd005: *core* ∈ Cores_of_Partition(*part*)
@grd006: processes_of_cores(*proc*) = *core*
@grd007: partition_mode(*part*) = PM_NORMAL
@grd008: process_state(*proc*) = PS_Ready ∨ process_state(*proc*) = PS_Running
@grd009: finished_core(*core*) = TRUE
...
@grd204: ∀p·p∈processes_of_partition~[{*part*}] ∧ p∈dom(currentpriority_of_process) ⇒
currentpriority_of_process(p) ≤ currentpriority_of_process(*proc*)
...
then
@act001: process_state := (process_state⩤(cores_processes(core) ↦ PS_Ready))⩤ {proc ↦ PS_Running}
@act002: current_processes(*core*) := *proc*
@act003: current_processes_flag(*core*) := TRUE
@act004: need_reschedule := FALSE
@act005: need_procresch(core) := FALSE
...

(1) If a running process is suspended by the service SUSPEND or SUSPEND_SELF, the state transition of this process is from *running* to *suspended*.

(2) If a running process requests a currently unavailable resource, such as events or mutexes occupied by other processes, the state transition is from *running* to *waiting*.

(3) If a process waiting for a resource is suspended by other processes, the state transition is from *suspended* to *WaitandSuspend*.

Based on the above analysis, the process states are defined as *partition(PROCESS_STATES, PS_Dormant, PS_Ready, PS_Waiting, PS_suspended, PS_Waitandsuspended, PS_Running, PS_Faulted)* in the context *C_Part_Proc_Trans*. Similar to the partition-mode-transition event *partition_mode_transition* at the top machine, the state transitions defined in the event *process_state_transition* are not concrete transition events or actions; they are only the possible state transition paths. The concrete services in the following machines, such as *SET_PRIORITY*, *SUSPEND*, *RESUME*, and *STOP*, will refine the concrete state transition paths.

5.2.2 Process Scheduling. One of the most important activities of the OS is to arbitrate the competition within a partition when multiple processes of a partition concurrently issue requests of processor cores.

The event of process scheduling in the bottom machine is listed in Table 11. In Table 11, the *grd005* determines whether the operating *core* is assigned to the partition *part*, and the *grd006* determines whether the core has been assigned to the process *proc* in the PCA. The *grd007* and the *grd008* confirm the current partition mode is NORMAL and the scheduled process is READY or RUNNING. The *grd009* examines whether the current processor *core* has finished the executions of prior services. The *grd204* can ensure the scheduled process *proc* is a process with the highest priority.

In the action part in Table 11, the *act002* and the *act003* assign the processor core *core* to the process *proc* and enable the process execution on this core. Meanwhile, the *act001* updates states for all processes in the partition.

5.2.3 Process Services. Processes may be designed as periodic or aperiodic executions. The periodic types are introduced in the third machine `M_Part_Proc_With_Events` and defined as *partition(PROC_PERIOD_TYPE, PERIOD_PROC, APERIOD_PROC)*. Three types of priority (*current*, *base*,

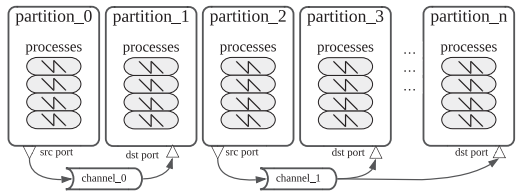

Fig. 10. Channels between interpartition communication ports.

and *retained*) are represented as variables *basepriority_of_process, currentpriority_of_process, retainedpriority_of_process* in the fourth machine, respectively.

We have listed an example of process service, CREATE_PROCESS, in Section 3.6. This service is divided into four subprocedures, and each procedure is specified as one event that models the entry, the intermediate, and the last action behaviors, respectively.

5.3 Inter- and Intrapartition Communications

ARINC 653 defines two types of communication paradigms: interpartition communications and intrapartition communications. Interpartition communications refer to communications between partitions, while intrapartition communications refer to communications between constituent processes within a partition.

Interpartition communications are composed of two modes: the queuing and sampling. The queuing involves storing messages in a queue until they are read by the receiving partition, while the sampling involves periodically checking for the availability of a new message.

Intrapartition communications can take place in several modes: blackboards, buffers, mutexes, events, and semaphores. Blackboards and buffers provide general interprocess message communications and synchronization, while semaphores, events, and mutexes provide only process synchronizations.

In ARINC 653, messages are atomic entities. We use the set *MESSAGES* in `Ctr_IPC` to represent the set of messages, and the variable *used_messages* to represent the set of already sent (used) messages. When sending messages in interpartition communications, the messages to be sent should be in the set *MESSAGES*, and they are stored in *used_messages* after being sent.

5.3.1 Interpartition Communications. The significant difference between inter- and intrapartition communications is the channel mechanism for interpartition communications. A channel serves as a link between a source port and one or more destination ports, illustrated in Figure 10. Ports allow a partition to send or receive messages on a specific channel. We use two types of channel events, *transfer_sampling_msg* and *transfer_queuing_msg*, to model channel behaviors for sampling and queueing modes, respectively.

In sampling channels, "*a message remains in the source port until it is transmitted by the channel or it is overwritten by a new occurrence of the message, whichever occurs first.*" On the other hand, in queuing channels, "*a message sent by the source partition is stored in the message queue of the source port until it is transmitted by the channel. When the message reaches the destination port, it is stored in a message queue until it is received by the destination partition.*"

We use *partition (PORTS, SamplingPorts, QueuingPorts)* to represent interpartition communication ports, and $Queuing\_Channels \in Source\_QueuingPorts \rightarrowtail\!\!\!\!\!\!\rightarrow Dest\_QueuingPorts$ to represent queuing channels. A port has a message space to store the message(s) to be sent/received. Sampling ports have a single message storage, which is specified as a variable $msgspace\_of\_samplingports \in sampling\_ports \nrightarrow (MESSAGES \times \mathbb{N})$. Queuing ports keep a message queue with a queuing length,

Table 12. The Event *send_queuing_message*

send_queuing_message $\widehat{=}$ **any** *core port msg t part*
where
...
@grd005: finite(queue_of_queuingports(*port*)) ∧
card(queue_of_queuingports(*port*)) <
MaxMsgNum_of_QueuingPorts(*port*)
@grd006: processes_waitingfor_queuingports(*port*) =∅
...
@grd201: *part* = current_partition
@grd202: Ports_of_Partition(*port*) = *part*
@grd203: t = clock_tick×ONE_TICK_TIME
then
@act001: queue_of_queuingports(*port*) :=
queue_of_queuingports(*port*) ⩤ {*msg* ↦*t*}
@act002: used_messages := used_messages ∪ {*msg*}

Table 13. Insert Waiting Processes into the READY Queue in Service *DISPLAY_BLACKBOARD*

DISPLAY_BLACKBOARD $\widehat{=}$ **any** *part procs core bb msg procs_Ready*
@grd006: partition_mode(*part*) = PM_NORMAL
@grd012: *procs* = processes_waitingfor_blackboards(*bb*)
@grd014: location_of_service3(*core*) =
Display_Blackboard_NeedWakeup↦loc_1
@grd015: procs_Ready ∈ procs → {PS_Ready}
then
@act001: location_of_service3(*core*) :=
Display_Blackboard_NeedWakeup↦loc_2
@act002: msgspace_of_blackboards(*bb*) := *msg*
@act003: processes_waitingfor_blackboards(*bb*) :=
processes_waitingfor_blackboards(*bb*) \ *procs*
@act004: used_messages := used_messages∪{msg}
@act005: process_state := process_state⩤procs_Ready

which is specified as a variable *queue_of_queuingports* ∈ *queuing_ports* → *(MESSAGES* × ℕ*)*. The pair *(MESSAGES* × $\mathbb{N}_1$*)* represents a message sent/received at a specific time. For example, writing a sampling message *msg* on the port *pt* at time *t* is represented as *pt* ↦ *(msg* ↦ *t)* ∈ *msgspace_of_samplingports*. The queuing mode is queued in the FIFO or priority order. We use a variable *processes_waitingfor_queuingports* ∈ *queuing_ports* → *(processes* ⇸ *(MESSAGES*×ℕ*))* to store waiting processes of a queuing port.

The event *send_queuing_message* specifies the service *SEND_QUEUING_MESSAGE*, which is illustrated in Table 12. This event involves sending a message *msg* via a queuing port *port* when the port is not full and no other processes are waiting for it. The guard *grd05* checks whether the current number of the queue port is less than the queue size. The guard *grd06* checks whether other processes are waiting for this port. The sending behavior is defined as *act01*, and the message is set to *used* state after being sent by the action *act02*.

5.3.2 Intrapartition Communications. Intrapartition communications provide provisions for processes within a partition to communicate with each other.

Buffers can store multiple messages in a message queue. When using buffers, no messages will be lost, and the sender will be blocked if the queue is full. The size of a message queue is specified as ∀*buf.(buf*∈*buffers* ∧ *finite(queue_of_buffers(buf))* ⇒ *card(queue_of_buffers(buf))* ≤ *MaxMsgNum_of_Buffers(buf))*.

No message queuing occurs in **blackboards**. Any message written to a blackboard remains invariant until the message is either cleared or overwritten. Processes waiting for blackboards are specified as *processes_waitingfor_blackboards* ∈ *blackboards* → ℙ*(processes)*.

Semaphores provide controlled accesses to partition resources, and they can be counted. Processes waiting for semaphores are specified as *processes_waitingfor_semaphores* ∈ *semaphores* → *(processes* ⇸ ℕ*)*.

An **event** contains a bool-state variable, namely "up" and "down," and a waiting-process queue. The bool-state variable of an event is specified as *partition(EVENT_STATE, {EVENT_UP}, {EVENT_DOWN})*, and the processes waiting for events are specified as *processes_waitingfor_events* ∈ *events* → ℙ*(processes)*.

A **mutex** is a synchronization object to control access to partition resources. Only one process at a time can own a mutex, which is specified as *partition(MUTEX_STATE, MUTEX_AVAILABLE, MUTEX_OWNED)*. The process owning a mutex is specified as *mutex_of_process* ∈ *mutexs* ⇸ *processes*. A mutex contains a defined priority value that a process's current priority is raised to when the mutex is obtained by this process. Processor cores that request mutexes are specified as *acquire_mutex* ∈ *CORES* ⇸ *mutexs*.

Table 14. The Creation of the Error Handler

CREATE_ERROR_HANDLER $\widehat{=}$ **any** *part proc core* **where**
@grd004: location_of_service(*core*) = Create_Process↦loc_i
@grd005: finished_core(*core*) = FALSE
@grd009: partition_mode(*part*) = PM_COLD_START ∨
partition_mode(*part*) = PM_WARM_START
...
then
@act001:
location_of_service(*core*) := Create_Process↦loc_1
@act002: process_state(proc) := PS_Dormant

Table 15. Project Statistics

Machine	Events	LOC	POs	AD
I	1	21	6	6(100%)
II	16	382	240	233(97%)
III	31	670	232	211(90%)
IV	110	3,000	2,055	1810(88%)
V	242	7,200	981	823(93%)
VI	257	7,800	164	164(100%)
VII	276	8,760	136	134(98%)
Total	-	-	3,814	3,471(91%)

The event *display_blackboard_needwakeuprdprocs_insert* in the fifth-layer machine, as illustrated in Table 13, specifies the action "*remove the processes from the blackboard's process queue and move them from the WAITING to the READY state*" in the *DISPLAY_BLACKBOARD* service. This event means that if the partition is in NORMAL (*grd006*), the processes *procs* are waiting for the blackboard *bb* (*grd012*), and the service has executed the procedure 1 (*grd014*), then the service will transfer the message *msg* on the blackboard *bb* (*act002, act003, act004*), set the service execution location (*act001*), and set the WAITING processes to READY and insert them into the READY queue (*act005*).

5.4 Health Monitoring

Health monitoring (HM) functions respond to and report hardware, application, and/or OS software errors and failures. The HM mechanism is introduced in the last machine `M_HM`. The ARINC 653 OS will provide HM configuration tables and an application-level error handler process for each partition. The error handler is a special process in a partition with the highest priority but no process identifiers. It defines a variable *shutdown* ∈ *BOOL* to control a module and refines each event in the machine `M_IPC` by adding a guard *shutdown = FALSE*. This means that if a module is shut down, no event can be triggered. Each partition having at most one error handler process is specified as a partial function *errorhandler_ of_partition* ∈ *PARTITIONS* ⤔ *processes*.

The event *create_error_handler* in Table 14 in the `M_HM` specifies the action "*Create a special process*" in the service *CREATE_ERROR_HANDLER*. This event means that if the service has completed the initialization (grd004) and the partition state is in *COLD_START* or *WARM_START* (*grd009*), the event will set the current process to *Dormant* (*act002*) and the location at *loc_1* (*act001*).

5.5 Timing Managements

The event *ticktock* is used to represent the clock ticks. The variable *clock_tick* specifies the system time clock and the variable *need_reschedule* is a flag used to denote whether the system can trigger scheduling. Both *clock_tick* and *need_reschedule* will be used in process or partition events.

The timing management services provide a means to control periodic and aperiodic processes. At the end of each processing cycle, a periodic process can call the *PERIODIC_WAIT* service to get a new deadline. The *TIMED_WAIT* service allows the process to suspend itself for a minimum amount of elapsed time. After the wait time has elapsed, the process becomes available to be scheduled. These two services are first introduced as *period_wait* and *time_wait* in the third machine `M_PartProc_With_Events`.

We define a variable *timeout_trigger* ∈ *processes* ⇸ *PROCESS_STATES* × $\mathbb{N}_1$ to store the time counter for processes. The ordered pair *(proc* ↦ *(PS_Ready* ↦ *t))* ∈ *timeout_trigger* means that the process *proc* is blocked and is waiting for some unavailable resource until time *t*, and then this process will be set to *READY*. When the system initiates a time counter with a duration *t*, a tuple *(process_state* × *t)* will be inserted into the timeout trigger. When the time *t* comes, the event *time_out* triggers a blocked process to the state *process_state*.

6 PROOF OBLIGATIONS AND RESULTS

6.1 Project Evaluation

The statistics of our specification and proof are shown in Table 15. We can see that the **lines of code (LOC)** and events for machines increase gradually since the refinement machine is an extension of the above machine. The last-layer machine M_HM contains 276 events and 8,760 LOC. The number of invariants representing critical properties for all the machines is 157. More than 3,800 **proof obligations (POs)** are generated, and more than 90% of them are automatically discharged.

6.2 Proof Obligations

Proof obligations can ensure concrete machines hold the specified properties of the abstract machines. We have proved the most important types of proof obligations: gluing and consistency invariants, deadlock freedom, correct refinement relations, and so on. Most of the proof obligations can be proved by the SMT solver, the model checker proB [26], or other automatic provers.

6.2.1 Invariant Preservation in the First Layer. Invariant preservation in the first layer requires formal proof to ensure that various events maintain the invariants constrained in this layer. This proof process ensures that modified invariants hold true in the presence of defined constants and variables, which can be represented as "*axioms*, *invariants* ⊢ *modified invariants*." As a result, this type of proof obligation can be formally defined as "$A\ c, I\ c\ v \vdash I'\ c\ v'$."

6.2.2 Invariant Establishment Rules for the Initialization Event. The initialization event should *establish the invariant for the first time* [3]. In this way, other events after initialization can be enabled in a situation where the invariants hold. The initialization event establishes invariants that can be described as "*axioms* ⊢ *modified invariants by the initialization event*." This proof can be formally defined as "$A\ c \vdash I_i'\ c\ v_i'$," in which v_i' denotes the modified variables after executing the initialization event and I_i' denotes the modified invariants of the initialization event.

6.2.3 Consistency. Consistency proofs include feasibility statements (see Section 2.6 in [11]), consistency of hypotheses, and consistency among dynamic and static elements (see Section 8.1.5 in [4]).

Feasibility statements (FIS) shows that under the given properties (axioms), the invariants, and the guards, the event indeed yields at least one valid value as defined by the predicates in that event. This can be described as "*axioms*, *invariants*, *guards*, *predicates* ⊢ *new variables defined by predicates*," which can be formally defined as "$A\ c, G\ c\ v, I\ c\ v \vdash \exists v'.v' = E(c, v)$." For example, the action $act313 : current\_partition :\in PARTITIONS$ in the initialization event of the fourth-level machine needs an FIS proof that indicates that the initialization event can generate a valid partition with the axiom constraint "axm_part_num: card(PARTITIONS) > 0 ∧ card(PARTITIONS) < 256."

Consistency of hypotheses refers to the requirement that the hypotheses, which are defined as axioms or invariants specified in machines and contexts, should not contradict each other. It is a logical statement that should hold true throughout the execution of the system. The proof obligations for consistency of hypotheses are generated automatically and labeled as INV.

Consistency among dynamic and static elements indicates that the model is coherent. It is used to prove the validation of event behaviors and state transitions in our model. These consistency conditions, defined as invariants, have been manually constructed, and their proof obligations have been proved in our work.

We use two examples to illustrate the consistency among dynamic and static elements. In the first example in Table 9, the event *set_part_mode2Normal_ready* is an intermediate event of the service *SET_PARTITION_MODE*, i.e., not the last and return event. There is a consistency invariant

Table 16. Gluing Invariants in the Event *partition_modetransition_to_warmstart*

partition_modetransition_to_warmstart $\widehat{=}$ **refines** partition_mode_transition **any** *part new procs cores* where
@grd001: part $\in$ PARTITIONS
@grd002: newm $\in$ PARTITION_MODES
@grd101: cores $\in \mathbb{P}_1$ (CORES)
@grd102: newm = PM_WARM_START
@grd103: partition_mode(part) = PM_WARM_START $\vee$ partition_mode(part) = PM_NORMAL
@grd104: procs = processes_of_partition^{-1}[{part}]
@grd105: cores = Cores_of_Partition(part)
then
@act001: partition_mode(part) := newm
@act101: processes := processes $\setminus$ procs
@act102: process_state := procs $\lhd$ process_state
@act103: processes_of_partition := procs $\lhd$ processes_of_partition
@act104: processes_of_cores := procs $\lhd$ processes_of_cores

that if an event of a service is not the last and return event of a service, the flag *finished_core(core)* that denotes that the core has finished a service should be FALSE. This is specified as an invariant, inv_local_service_and_finished_core, in the second layer, as follows:

$\forall$core,serv $\cdot$ core$\in$ dom(location_of_service) $\wedge$ serv $\in$ Services $\wedge$ location_of_service(core)$\neq$(serv $\mapsto$ loc_r) $\Rightarrow$ finished_core(core) = FALSE.

A another example, in Table 10, we define the abstract process-state transitions. In order to ensure that only processes in the NORMAL-mode partition can be at the RUNNING or READY state, we added an invariant in the model of the second layer. This invariant, *inv_runreadysuspfaltproc_onlyin_normal*, is specified as follows:

$\forall$part $\cdot$ { part$\in$ PARTITIONS $\wedge$ partition_mode(part) $\neq$ PM_NORMAL $\Rightarrow$
$\forall$proc $\cdot$ [proc$\in$ (processes_of_partition^{-1}[{part}]$\cap$dom(process_state)) $\wedge$ process_state(proc)$\in$ PROCESS_STATES$\Rightarrow$
process_state(proc) $\neq$ PS_Ready $\wedge$ process_state(proc) $\neq$ PS_Running $\wedge$
process_state(proc) $\neq$ PS_Suspend $\wedge$ process_state(proc) $\neq$ PS_Faulted] }.

6.2.4 Gluing Invariants. Gluing invariants are invariants that define specific relationships among variables and states between the concrete and abstract machines.

Gluing invariants can establish a rigorous link between the adjacent machines. They can include variables from the refined machine. Gluing invariants "glue" the space of the refined machine to its above abstract machine.

The event in Table 16 is refined from the abstract event that defines partition-mode transitions in Table 6. In Table 16, we added two parameters, the *procs* and *cores*, to represent the processes in a partition and the assigned cores to this partition, respectively. The gluing invariants are listed as guards *grd101* through *grd106* in Table 16. For example, the guard "grd104*: procs = processes_of_partition*$^{-1}$*[{part}]*" links the new argument *procs* to its partition *part* and denotes that *procs* is the set of all the processes in the partition *part*. The "*processes_of_partition* $\in$ *processes* $\nrightarrow$ *PARTITIONS*" denotes the mapping relation from a process to its partition. The guard "grd105*: cores = Cores_of_Partition(part)*" links the new arguments *cores* to their partition and denotes that the *cores* are the set of all processor cores assigned to the partition *part*. The proof obligations of these gluing invariants are listed in Table 12.

6.2.5 Deadlock Freedom. Deadlock freedom means a system will work forever once it is started [3]. This property confirms that the system can evolve by engaging at least one of its events at any time, which ensure that at least one of the guards of various events is always true. The deadlock freedom is described as "*axioms, invariants* $\vdash$ *disjunction of the guards,*" which is formalized by "$A\ c, I\ c\ v \vdash G_1\ c\ v \vee G_2\ c\ v \vee \ldots G_n\ c\ v$," in which $G_i\ c\ v$ denotes the guards for a specific event *i*. We use the model checker *ProB* [26], which can be installed into Rodin as a plug-in to prove the deadlock freedom.

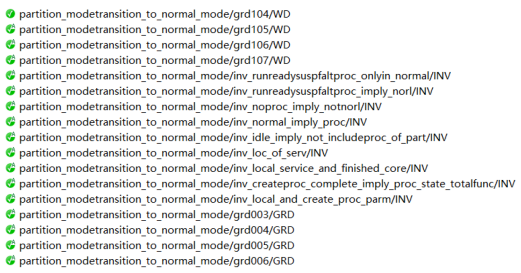

Fig. 11. Proof obligations for the event *partition_modetransition_to_normal_mode*.

Fig. 12. Proof obligations for the event *partition_modetransition_to_warmstart*.

6.2.6 Correct Refinement. To ensure correct refinements, we should prove two parts: the guard strengthening concerning the guards and correct refinement relations concerning the actions [3].

Guard strengthening is to prove the guards of a concrete event are stronger than those of an corresponding abstract event. The term "stronger" means that the former guards imply the later guards, which ensures an enabled concrete event transition without a counterpart to its corresponding abstract transition. We prove the guard strengthening as follows:

"*axioms, abstract invariants, concrete invariants, concrete guards* ⊢ *abstract guards.*"

This is formalized as "$A\ c,\ I\ c\ v,\ \mathcal{J}\ c\ v\ w,\ H_i\ c\ w \vdash G_i\ c,\ v$."

Correct refinement relations can ensure that the concrete model preserves the properties of the abstract model (i.e., the abstract invariants) and satisfies the added properties (i.e., the added invariants), while the guard strengthening guarantees only that the enabling conditions of a concrete event do not conflict with its corresponding abstract event. We prove the correct refinement as follows:

"*axioms, abstract invariants, concrete invariants, concrete guards* ⊢ *modified concrete invariant.*"

This is formalized as "$A\ c,\ I\ c\ v,\ \mathcal{J}\ c\ v\ w,\ H\ c\ w \vdash \mathcal{J}\ c\ v'\ w'$."

We illustrate two examples that demonstrate proof obligations of partial partition-mode transitions in Figures 11 and 12, respectively. The abstract partition-mode transitions are defined in the first layer and refined by *partition_modetransition_to_warmstart*, *partition_modetransition_to_normal_mode*, and a total of seven events in the second layer. Figure 11 displays the proof obligations of the event *partition_modetransition_to_normal_mode,* and Figure 12 shows those of the event *partition_modetransition_to_warmstart.*

Additionally, we have listed some important safety properties in Table 17. We refer readers to [3] for further details about the policy and concept of proof obligations.

6.3 Analysis

After a formal analysis of the Event-B specification, we confirm that the six errors found in the single-core ARINC 653 also exist in the multi-core ARINC653. The six errors, discussed in the literature [44], occur in process-state transitions in the process management services *RESUME* and *DELAYED_START*, and the communication services *SEND_QUEUING_MESSAGE* and *RECEIVE_BUFFER*. The reason is that the multi-core ARINC 653 derives from the single-core one, which contains the errors, so the multi-core ARINC 653 inherits these errors. We can address these errors using Event-B because we decompose each service into independent Event-B events that execute atomically and consider the locking mechanism for event executions, and the formal deductive analysis of the formal specification can uncover hiding problems through each refinement layer.

Table 17. Safety Properties for ARINC 653 Part 1–4

No.	Functionality/Invariant description
	Partition and process management
(1)	Each partition has at least one processor core.
(2)	A partition in COLD_START or WARM_START should not schedule a user application.
(3)	Each process has a core affinity.
(4)	Each process should only be scheduled on the core assigned to its partition.
(5)	A periodic process has a periodic release point.
(6)	A partition has a lock-level number.
(7)	Process management services should conform to the process state transition relations.
(8)	Partition management services should conform to the partition mode transition relations.
(9)	If the current lock level is greater than 0, the current process should not be scheduled out.
(10)	Each partition can have one health monitor process.
(11)	The scheduler is two-level scheduling that comprises partition scheduling and process scheduling.
(12)	The partition scheduling is static.
(13)	The process scheduling is dynamic, which picks the highest-priority process to execute on its assigned core.
(14)	If a partition is not in NORMAL, its applications should not be in Ready, Running, Suspend, or Faulted.
(15)	If a partition contains processes at Ready, Running, Faulted, or Suspend, the partition should be NORMAL.
(16)	There is at most one Running process in each processor core.
(17)	If a partition is in IDLE, it should not have any process.
(18)	If a partition is in NORMAL, it should have at least one process.
(19)	There is at most one Running process in each processor core.
(20)	When a partition is in COLD_START or WARM_START, the lock level should be larger than zero.
(21)	If the lock level of a partition is zero, the partition should be in NORMAL.
(22)	The current process's partition is in NORMAL.
	Communication
(23)	The sizes of message queues are finite.
(24)	The message number in a queue is less than the maximum message number.
(25)	Wake up all the waiting processes for events and blackboards when these resources become available again.
(26)	Wake up all the first waiting processes for ports, mutexes, etc., when these resources become available again.
(27)	Interpartition communication ports, events, buffers, and semaphores have process queuing discipline.
	Multi-core framework
(28)	Service can be interrupted by other services during execution.
(29)	Services can execute concurrently within a partition or between partitions.
(30)	The execution order of service should remain the same in the refinement.

7 RELATED WORK

The seL4 [2, 24, 25] is the first and most comprehensive OS kernel that uses formal methods to ensure safety and security properties of a high-performance L4-family microkernel [20]. However, seL4 does not support multicore concurrency with fine-grained interleaving and locking. For this issue, von Tessin [41] and Peters et al. [32] argue that concurrent behaviors executing on multicores can be reduced to a minimum, so they believe a single **big-kernel-lock (BLK)** might be good enough to achieve good performance on multicore platforms. Von Tessin further shows how to convert single-core seL4 proofs into proofs of a multicore OS kernel with a BKL framework.

Zhong Shao et al. present a compositional approach and build a clean-slate and certified concurrent OS kernel—CertikOS (Certified Kit Operating System) [1, 17–19]. Their concurrency [18] framework allows interleaved executions of kernel/user modules across different abstraction layers, and each such layer can have a different set of observable events and features [19]. They introduce the hypervisor architecture of the CertikOS that leverages formal certification to ensure information leakage in cloud computing [17]. CertiKOS can isolate applications from each other as well as provider-controlled resource management mechanisms.

Nevertheless, the well-known contributions of the above literatures focus on traditional OSs but not ARINC 653-compliant OSs.

Formal specification of the ARINC 653 architecture and components has been developed using Circus language [31], **Architecture Analysis and Design Language (AADL)** [14, 36, 43], and PROMELA of the SPIN [22] model checker [12]. However, they only specify a small part of services and analysis of few critical properties. Zhao et al. [34, 45] specify and verify interpartition communication of ARINC 653 in sequential and concurrent settings, respectively. They also redevelop and automatically verify ARINC 653 models using Python [35]. In [44], the system functionalities and all service requirements of ARINC 653 have been formalized in Event-B, and some errors have been found in the standard.

About the validation of the Event-B model, Ait-Ameur et al. [7] claim that the ontology is a good candidate for handling explicit domain knowledge, Mendil et al. [27] present a framework that facilitates the formalization of standard concepts and rules as an ontology using Event-B consisting of data types and a collection of operators and properties. At-Ameur et al. propose [6] to define transitions explicitly as partial functions in an Event-B theory. Zhao et al. [34, 45] also propose an ontology methodology based on Event-B as an intermediate model between informal descriptions of ARINC 653 and the formal specification in Event-B. However, all the above contributions support only the single-core ARINC 653.

8 CONCLUSION AND FUTURE WORK

This article proposes a formal specification method for the multi-core ARINC 653 standard using Event-B and its IDE RODIN, presents a comprehensive specification with seven refinement layers of machines, and performs a formal analysis on proof obligations. After formal specification and analysis, we confirm that the errors found in the single-core ARINC 653 also exist in the multi-core. In the following work, we will consider security properties in our current safety specification and organize the refinement structure to satisfy specific safety- and security-critical properties.

APPENDIX

For definition and description of proof obligations, we adopt the notations in Table 18 referencing from [3].

Table 18. Notations

Symbols	Description
c	constants
v	abstract variables
w	concrete variables
E	abstract events
F	concrete events
$v' = E\ (c,\ v)$	modified abstract variables after executing abstract events
$w' = F\ (c,\ w)$	modified concrete variables after executing concrete events
$A\ c$	axioms for constants
$G\ c\ v$	guards for abstract events
$H\ c\ w$	guards for concrete events
$I\ c\ v$	abstract invariants for constants and abstract variables
$\mathcal{J}\ c\ v\ w$	concrete invariants for constants and concrete variables
$I'\ c\ v'$	modified abstract invariants for constants and modified abstract variables
$J'\ c\ w'$	modified concrete invariants for constants and modified concrete variables

Meanwhile, we list some important operators with their mathematical semantics in Table 19.

Table 19. Glossary

Operator	Description	Formal Semantic
$\mathbb{N}$	The set of natural numbers	
$\setminus$	Set difference	$S \setminus T = \{ x \mid x \in S \wedge x \notin T\}$
$\mathbb{N}_1$	The set of positive natural numbers	$\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$
$\mathbb{Z}$	The set of integer numbers	
$\mathbb{P}$	Powerset	$\mathbb{P}(S) = \{s \mid s \subseteq S\}$
$\mathbb{P}_1$	Non-empty subsets	$\mathbb{P}_1(S) = \mathbb{P}(S) \setminus \{\emptyset\}$
$\leftrightarrow$	Relations	$S \leftrightarrow T = \mathbb{P}(S \times T)$
dom	Domain	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow \mathrm{dom}(r) = \{x \cdot (\exists y \cdot x \mapsto y \in r)\}$
ran	Range	$\forall r \cdot r \in S \leftrightarrow T \Rightarrow \mathrm{ran}(r) = \{y \cdot (\exists x \cdot x \mapsto y \in r)\}$
$\nleftrightarrow$	Total relations	if $r \in S \nleftrightarrow T$ then $\mathrm{dom}(r) = S$
$\leftrightarrow\!\!\!\!\rightarrow$	Surjective relations	if $r \in S \leftrightarrow\!\!\!\!\rightarrow T$ then $\mathrm{ran}(r) = T$
$\nleftrightarrow\!\!\!\!\rightarrow$	Total surjective relations	if $r \in S \nleftrightarrow\!\!\!\!\rightarrow T$ then $\mathrm{dom}(r) = S$ and $\mathrm{ran}(r) = T$
;	Forward composition	$\forall p,q \cdot p \in S \leftrightarrow T \wedge q \in T \leftrightarrow Q \Rightarrow p;q = \{x \mapsto y \mid \exists z \cdot x \mapsto z \in p \wedge z \mapsto y \in q\}$
$\circ$	Backward composition	$p \circ q = q ; p$
$\lhd$	Domain restrictions	$S \lhd r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \in S\}$
id	Identity	$S \lhd id = \{x \circ x \mid x \in S\}$
$\lhd\!\!\!\!-$	Domain subtraction	$S \lhd\!\!\!\!- r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\}$
$\rhd$	Range restriction	$r \rhd T = \{x \mapsto y \mid x \mapsto y \in r \wedge y \in T\}$
$\rhd\!\!\!\!-$	Range subtraction	$r \rhd\!\!\!\!- T = \{x \mapsto y \mid x \mapsto y \in r \wedge y \notin T\}$
r^{-1}	Inverse	$r^{-1} = \{y \mapsto x \mid x \mapsto y \in r\}$
$r[S]$	Relational image	$r[S] = \{y \mid \exists x \cdot x \in S \wedge x \mapsto y \in r\}$
$\lhd\!\!\!\!+$	Overriding	$r_1 \lhd\!\!\!\!+ r_2 = r_2 \cup (\mathrm{dom}(r_2) \lhd\!\!\!\!- r_1)$
$\otimes$	Direct product	$p \otimes q = \{x \mapsto (y \mapsto y) \mid x \mapsto y \in p \wedge y \mapsto z \in q\}$
$\nrightarrow$	Partial functions	$S \nrightarrow T = \{r \cdot r \in S \leftrightarrow T \wedge r^{-1};r \subseteq T \lhd id\}$
$\rightarrow$	Total functions	$S \rightarrow T = \{f \cdot \in S \nrightarrow T \wedge \mathrm{dom}(f) = S\}$
$\rightarrowtail$	Partial injections	$S \rightarrowtail T = \{f \cdot \in S \nrightarrow T \wedge f^{-1} \in T \nrightarrow S\}$
$\rightarrowtail$	Total injections	$S \rightarrowtail T = S \rightarrowtail T \cap S \rightarrow T$
$\twoheadrightarrow$	Partial surjections	$S \twoheadrightarrow T = \{f \cdot \in S \nrightarrow T \wedge \mathrm{ran}(f) = T\}$
$\twoheadrightarrow$	Total surjections	$S \twoheadrightarrow T = \{f \cdot \in S \twoheadrightarrow T \wedge S \rightarrow T\}$
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	Bijections	$S \rightarrowtail\!\!\!\!\twoheadrightarrow T = \{f \cdot \in S \rightarrowtail T \wedge S \twoheadrightarrow T\}$

REFERENCES

[1] n. d. *CertiKOS*. http://flint.cs.yale.edu/certikos/index.html/

[2] n. d. *seL4*. https://sel4.systems/

[3] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[4] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: An open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12, 6 (2010), 447–466.

[5] SAE-ITC Aeronautical Radio, Inc. December 23, 2019. *ARINC Specification 653 Part 1: Avionics Application Software Standard Interface, Part 1 - Required Services*.

[6] Yamine Aït-Ameur, Guillaume Dupont, Ismail Mendil, Dominique Méry, Marc Pantel, Peter Rivière, and Neeraj K. Singh. 2022. Empowering the event-B method using external theories. In *Integrated Formal Methods: 17th International Conference (IFM'22), Proceedings*. Springer, 18–35.

[7] Yamine Ait-Ameur and Dominique Méry. 2016. Making explicit domain knowledge in formal system development. *Science of Computer Programming* 121 (2016), 100–127.

[8] June Andronick. 2017. Reasoning about concurrency in high-assurance, high-performance software systems. In *International Conference on Automated Deduction*. Springer, 1–7.

[9] June Andronick. 2018. *Formal Model of a Multi-core Kernel-based System*. Technical Report. National ICT Australia Limited.

[10] Pedro de la Cámara, María del Mar Gallardo, and Pedro Merino. 2007. Model extraction for Arinc 653 based avionics software. In *International SPIN Workshop on Model Checking of Software*. Springer, 243–262.

[11] Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna. 2005. RODIN (rigorous open development environment for complex systems). *School of Computing Science Technical Report Series*.

[12] Pedro de la Cámara, J. Raúl Castro, María del Mar Gallardo, and Pedro Merino. 2011. Verification support for ARINC-653-based avionics software. *Software Testing, Verification and Reliability* 21, 4 (January 2011), 267–298.

[13] Julien Delange and Laurent Lec. 2011. POK, an ARINC653-compliant operating system released under the BSD license. In *13th Real-Time Linux Workshop*, Vol. 10. 181–192.

[14] Julien Delange, Laurent Pautet, and Fabrice Kordon. 2010. Modeling and validation of ARINC653 architectures. In *Proceedings of Embedded Real Time Software and Systems (ERTS'10)*.
[15] The Coq Development Team. n. d. *The Coq Proof Assistant*. https://coq.inria.fr/
[16] Kevin Elphinstone, Amirreza Zarrabi, Adrian Danis, Yanyan Shen, and Gernot Heiser. 2016. An evaluation of coarse-grained locking for multicore microkernels. *arXiv preprint arXiv:1609.08372* (2016).
[17] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A certified kernel for secure cloud computing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*. 1–5.
[18] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Communications of the ACM* 62, 10 (2019), 89–99.
[19] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 653–669.
[20] Jorrit N Herder. 2005. *Towards a True Microkernel Operating System*. Master's Thesis, Vrije Universiteit Amsterdam 2005.
[21] Green Hills. n. d. *INTEGRITY-178 tuMP*. https://www.ghs.com/products/safety_critical/integrity_178_tump.html
[22] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295.
[23] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. 2013. Developing mode-rich satellite software by refinement in event-B. *Science of Computer Programming* 78, 7 (2013), 884–905.
[24] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 1–70.
[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock1, Philip Derrin1, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 207–220.
[26] Michael Leuschel and Michael Butler. 2003. ProB: A model checker for B. In *Formal Methods: International Symposium of Formal Methods Europe (FME'03), Proceedings*. Springer, 855–874.
[27] Ismail Mendil, Yamine Aït-Ameur, Neeraj Kumar Singh, Dominique Méry, and Philippe Palanque. 2021. Standard conformance-by-construction with Event-B. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 126–146.
[28] Dominique Méry and Neeraj Kumar Singh. 2013. Formal specification of medical systems by proof-based refinement. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 1 (2013), 1–25.
[29] University of Cambridge and Technische Universität Münch. n. d. *Isabelle/HOL*. https://isabelle.in.tum.de/
[30] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1 (2007), 271–307.
[31] Artur Oliveira Gomes. 2012. *Formal Specification of the ARINC 653 Architecture Using Circus*. Ph. D. Dissertation. University of York.
[32] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*. 1–7.
[33] John Rushby. 1999. *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance*. Technical Report.
[34] David Sanan, Yongwang Zhao, Shang-Wei Lin, and Liu Yang. 2021. CSim2: Compositional top-down verification of concurrent systems using rely-guarantee. *ACM Transactions on Programming Languages and Systems* 43, 1, Article 2 (Feb. 2021), 46 pages. https://doi.org/10.1145/3436808
[35] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 287–305.
[36] Frank Singhoff and Alain Plantec. 2007. AADL modeling and analysis of hierarchical schedulers. In *Proceedings of the 2007 ACM International Conference on Sigada Annual International Conference*. 41–50.
[37] Wind River System. n. d. *VxWroks 653*. https://resources.windriver.com/vxworks/vxworks-653-product-overview
[38] Lynx Software Technologies. n. d. *LynxOS-178*. https://www.lynx.com/products/lynxos-178-do-178c-certified-native-posix-partitioned-rtos-more-info
[39] Freek Verbeek, Oto Havle, Julien Schmaltz, Sergey Tverdyshev, Holger Blasum, Bruno Langenstein, Werner Stephan, Burkhart Wolff, and Yakoub Nemouchi. 2015. Formal API specification of the PikeOS separation kernel. In *NASA Formal Methods Symposium*. Springer, 375–389.
[40] Michael Von Tessin. 2012. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures*.

[41] Michael von Tessin. 2013. *The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-system Kernels*. Ph. D. Dissertation. University of New South Wales, Sydney, Australia.

[42] Ying Wang, Dianfu Ma, Yongwang Zhao, Lu Zou, and Xianqi Zhao. 2011. An AADL-based modeling method for arinc653-based avionics software. In *2011 IEEE 35th Annual Computer Software and Applications Conference*. IEEE, 224–229.

[43] Ying Wang, Dianfu Ma, Yongwang Zhao, Lu Zou, and Xianqi Zhao. 2011. An AADL-based modeling method for ARINC653-based avionics software. In *Proceedings of IEEE 35th Annual Computer Software and Applications Conference (COMPSAC'11)*. 224–229.

[44] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. 2016. Formal specification and analysis of partitioning operating systems by integrating ontology and refinement. *IEEE Transactions on Industrial Informatics* 12, 4 (2016), 1321–1331.

[45] Yongwang Zhao, David Sanan, Fuyuan Zhang, and Yang Liu. 2019. Refinement-based specification and security analysis of separation kernels. *IEEE Transactions on Dependable and Secure Computing* 16, 1 (2019), 127–141. https://doi.org/10.1109/TDSC.2017.2672983

[46] Yongwang Zhao, Zhibin Yang, David Sanán, and Yang Liu. 2015. Event-based formalization of safety-critical operating system standards: An experience report on ARINC 653 using Event-B. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. IEEE, 281–292.

Received 26 July 2022; revised 27 June 2023; accepted 10 August 2023