Abstract—Standards play the key role in safety-critical systems. Errors in standards could mislead system developer’s understanding and introduce bugs into system implementations. In this paper, we present an Event-B formalization and verification for the ARINC 653 standard, which provides a standardized interface between safety-critical real-time operating systems and application software, as well as a set of functionalities aimed to improve the safety and certification process of such safety-critical systems. The formalization is a complete model of ARINC 653, and provides a necessary foundation for the formal development and verification of ARINC 653 compliant operating systems and applications. Three hidden errors and three cases of incomplete specification were discovered from the verification using the Event-B formal reasoning approach.

I. Introduction

In recent years, safety-critical systems have paved the way for the integration on one single platform of different criticality level application subsystems developed by different vendors, like the Integrated Modular Avionics (IMA) [1] in avionics domain. IMA aims Partitioning Operating Systems (POS) implementation, supporting spatial and temporal partitioning [2]. Partitioning [1] provides independent execution of one or more applications, which procures temporal and spatial separation and fault containment to prevent propagation of application failures. In this way, partitioning is equivalent to an idealized system in which each partition allocates an independent processor and associated peripherals for the execution of applications, where all inter-partition communications are carried out on dedicated lines, giving applications with an environment which is undistinguishable from that provided by a physically distributed system.

In avionics industry, ARINC 653 [2] was first published in 1996 as a set of specifications to guide manufacturers in avionic application software towards maximum standardization. It aims to provide a standardized interface between a given POS and application software, as well as a set of functionalities to improve safety and certification process of safety-critical systems. ARINC 653 compliant POSs have been widely applied in safety-critical domains. Typical POSs are VxWorks 653 platform, INTEGRITY-178B, LynxOS-178, PikeOS, and open source software, e.g. POK [3] and Xtratum [4]. The ARINC 653 standard can be considered as a requirement for POSs under construction and a base for compliance tests on their implementation. However, hidden inconsistencies or incorrectness in standards could mislead system developers’ understanding, causing failures or malfunction in POSs, and hence a breakdown of applications, which is not allowed in safety critical systems. For this reason, ensuring standards correctness, and in particular ARINC 653 correctness, w.r.t. a set of functional, safety, and security properties, is necessary in ensuring absence of failures on safety-critical systems development.

Due to POSs’ complexity, traditional test-based techniques are not enough to warrant their correctness, as it is not possible to generate all necessary test cases to fully cover all behaviours of POSs, and hence to ascertain their correctness. During last decades, formal methods based techniques have been widely applied on the verification of both software and hardware [5]. In the field of real-time operating systems for safety-critical systems, most of related work has been concentrated on specifying or verifying the operating system [6]. Nevertheless, as the interface of the fundamental execution environment of IMA applications, a formal model of ARINC 653 is strongly necessary for formal development and verification of ARINC 653 compliant operating systems and ARINC 653 based applications. Although some research efforts have been paid off in the formal modelling and verification of ARINC 653 based systems ([7], [8], [9], [10], [11]), to our knowledge the formalization introduced in this work is the most complete model of the ARINC 653. We provide a detailed comparison with related work at Section II.

This paper presents the formalization of ARINC 653 Part 1 (the latest version, Version 3) [2], its verification using a deductive verification approach [12], and the errors in the standard found out during the verification. An ARINC 653 formal model is constructed using Event-B [13], a mathematical approach based on a model-driven design methodology used for specifying and reasoning about complex systems, including concurrent and reactive systems. Event-B uses the set theory as a modelling notation, refinement to represent systems at different abstraction levels, and mathematical proofs to verify consistency between refinement levels [13]. We choose Event-B due to the following reasons: (1) a specification in Event-B is easy to understand and has a strong development environment—Rodin, for which there exists many plugins to translate Event-B specifications into other formalization and source code, and for model visualization and simulation, among other functionalities; (2) its high degree of automatic reasoning eases the verification, and the inductive approach avoids state space explosion when verifying complicated systems; (3) events are very suitable for modelling operating systems, where hardware components, e.g. interrupts like clock and timers, need to be well managed.

We have modelled the system functionality of POS and all of 57 services specified in ARINC 653 Part 1, including partition and process management, time management, inter-
and intra-partition communication, and health monitoring. We use the deductive verification approach supported in Event-B and its Rodin platform [14] as a means for consistency checking. The description of the system functionality in ARINC 653 is manually formalized as the top level specification and safety properties (invariants), and the service requirements are translated into the low level specifications. Safety properties on the specifications and refinement between the top level and low level specifications are proven by discharging proof obligations. Finally, we found three errors in ARINC 653 Part 1, amongst which, one in service of process management, and two in services of inter- and intra-partition communication. Additionally we detected three cases where the specification of process state transitions is incomplete.

The rest of this paper is organized as follows. In Section II, we introduce the background and related work. Our framework approach is presented in Section III. The formalization of ARINC 653 is presented in Section IV. Section V presents the formalization and verification result, and a discussion. Finally, Section VI gives the conclusion and future work.

II. BACKGROUND AND RELATED WORK

A. Event-B

Here, we provide a brief overview of Event-B. Full details are provided in [15]. The Event-B method [13] is used to build reliably systems using discrete system models and aims at obtaining systems which can be considered to be correct by construction, in the sense that the systems produced are guaranteed to implement the corresponding functional specification.

Event-B models are described in terms of contexts and machines. Contexts specify the static part of a model whereas machines specify the dynamic part. Suppose a machine $M$, seeing a context $C$ with sets $s$ and constants $c$. An event of this machine is represented as

$$E \triangleq \text{any } x \text{ where } G(s, c, v, x)$$

$$\text{then } v : BA(s, c, v, x, v') \text{ end} \quad (1)$$

$E$ is the event name, $G(s, c, v, x)$ is the guard of the event that states the necessary condition for the event to occur, and $v : BA(s, c, v, x, v')$ is the action that defines how the state variables evolve when the event occurs. Actions use a before-after predicate, which relates the values $v$ (before the action) and $v'$ (afterwards). In a machine, the execution of an event is considered to take no time and no two events can occur simultaneously. When the guards of one or more events are true, one of these events necessarily occurs and the state is modified accordingly. Then, the guards are checked again, and so on.

Refinement in Event-B provides a means for introducing details about the dynamic properties of a model. A machine $RM$ can refine another machine $M$ and we call $M$ to the abstract machine (or refined machine), which specifies $RM$, and $RM$ the concrete machine (or refinement machine), which implements $M$.

Event-B defines proof obligations, which must be proven to show that machines hold the properties specified over them. For space reasons, we only describe here some of the most significant proof obligations. Formal definitions of all proof obligations are given in [15]. (1) Invariant preservation states that invariants are maintained whenever variables change their values. It ensures events preserve invariants specified over a machine. (2) Guard strengthening makes sure that the concrete guards in a concrete event are stronger than the abstract ones in the abstract event. This ensures that when a concrete event is enabled, so is the corresponding abstract one. (3) Simulation makes sure that each action in an abstract event is correctly simulated in the corresponding refinement. This ensures that when a concrete event is executed its actions are not contradictory with the actions in the corresponding abstract event.

Finally, the Rodin tool [14] is an Eclipse-based IDE supporting the application of the Event-B method. This is an industrial-strength tool for creating and analyzing Event-B models. It includes a proof-obligation generator and support for interactive and automated theorem proving.

B. ARINC 653

The current version of ARINC 653 defines a partitioning architecture for safety-critical systems on single-core as shown in Fig. 1. A POS is in fact a small partitioning kernel that provides operating system services according to the safety features required by the safety integrity level. The latest version of ARINC 653 published in 2010 is organized in six parts. Part 1, currently in Version 3, defines the standard APplication EXecutive (APEX) interface between the application software and the POS, and the list of services which allow the application software to control the scheduling, communication, and status information of its internal processing elements [2]. We focus our work in formalizing and verifying Part 1, since it specifies the baseline operating environment for application software used within IMA, and most of industrial and open source POSs implementing ARINC 653 are compliant with this part. ARINC 653 Part 1 concentrates on specifying the system functionality, which is described in natural language, and service requirements, which is presented by a type of pseudo-code: the APEX service specification grammar.

The required services specified in ARINC 653 Part 1 are grouped into the following major categories: partition management, defining partitions services, attributes, and the partition operating modes, the set of states a partition can be in and the transitions among them; process management, defining processes services, attributes, and the process operating modes; time management, defining time services for partitions and attributes; inter-partition communication, defining
procedure STOP
  (PROCESS_ID : in PROCESS_ID_TYPE;
   RETURN_CODE : out RETURN_CODE_TYPE) is
  error
    when (PROCESS_ID does not identify an existing process or identifies the current
          process) =>
      RETURN_CODE := INVALID_PARAM;
    when (the state of the specified process is DORMANT) =>
      RETURN_CODE := NO_ACTION;
  normal
    reset the specified process state to DORMANT;
    if (current process is error handler and PROCESS_ID is process which the error
        handler processed) then
      reset the partition’s LOCK_LEVEL counter (i.e., enable preemption);
    end if;
    if (specified process is waiting in a process queue) then
      remove the process from the process queue;
    end if;
    stop any time counters associated with the specified process;
    RETURN_CODE := NO_ERROR;
  end STOP;

Fig. 2. The service requirement of STOP service in process management.

communication modes between partitions, services provided, and attributes; intra-partition communication, similar to inter-
partion communication, but oriented to processes instead of
partitions; and health monitoring, which defines actuation rules
under system, partition, and application failures. Note that since
partitions, and therefore their associated memory spaces, are
defined during system configuration and initialization, there is
no memory allocation service defined in APEX. All required
services are specified in detail by pseudo-code in the service
specification grammar. For instance, the STOP service from
process management is illustrated in Fig. 2. For a detailed
description of partitions, processes, and services provided by
ARINC 653 we refer the reader to the ARINC 653 Standard [2].

C. Related work

A formal specification of the ARINC 653 architecture
using the Circus language is presented in [9]. Its specification
focuses on the whole ARINC 653 architecture and interactions
between IMA components. However, the formal model
only covers a small part of ARINC 653 services and no verification is carried out. Also focussing only in modelling,
ARINC 653 components and their constraints are modelled
using AADL (Architecture Analysis and Design Language)
that can be used for model-driven development of IMA application
([10], [11]), but not all APEX services are covered in these
works. In [8], only an ARINC 653 hierarchical scheduler is
modelled with AADL. Works in [7], [16] target not only the
ARINC specification but also its verification, where ARINC
653 services are modelled in PROMELA and verified using the
SPIN model checker to ensure the correctness of avionics
software constructed on top of ARINC 653. Here, the ARINC
and the application models, which are extracted from the
application’s C source code, comprise the complete formal
model for verification. However, the verification is focused
only on process and time management, not covering any other
ARINC 653 service or functionality.

The specification and verification of RTOSs and separation
kernels are also related to our work, as we overview in our
technical report [6]. It is worth noting that formal methods
have been largely used in the specification of separation kernels, a
generalization of POSs, like [17], [18], separation-partitioning
micro-kernels [19], or OSEK/VDX ([20], [21]), an international
standard for automotive operating systems. In general, formal
verification has been used on RTOSs for safety/security certification in industry, like in the AAMP7G microprocessor, which
is a hardware implementation of partitioning in Rockwell
Collins [22]; in PikeOS, which is an ARINC 653 compliant
POS in SYSGO AG ([23], [24]); in INTEGRITY-178B which is
also an ARINC 653 compliant POS in Green Hills [25]; and
in an Embedded Devices kernel in Naval Research Laboratory
[26]. It is worth to highlight the work in [27], where the
sel4 microkernel has been fully verified from the top level
specification down to the machine code.

The B-Method (predecessor of Event-B), has been also applied
to operating systems. It has been used for the (partial)
formal development of a secure partitioning kernel in the
Critical Software company [28]. A real-time operating system,
FreeRTOS, has also been formally specified using B method
[29]. The L4 microkernel, a kernel of general purpose OSs,
has also been formally modelled in B ([30], [31]). Event-B
extends B-Method with events, procedures that are activated
when a guard is enabled, and it is suitable for modelling
systems based on events. Both Event-B and B-Method share
the same foundation, the main difference between them is
that refinement in Event-B requires the guards of events of
a concrete implementation to be stronger than the guards of
the events in the abstract machine.

Formal verification on standards has attracted considerable
attentions for a long time, such as communication standard-
s/protocols ([32], [33], [34], [35]). It has also been taken into
account in safety-critical systems. Typical verification about
standards in this domain is the verification of compliance to
safety standards, such as in [36].

III. FRAMEWORK APPROACH

We first present our framework approach in this section.

Modeling consideration. The first aspect to be considered
is what to be modelled. The complete document structure
and the number of pages of each section of ARINC 653 Part 1
are as shown in the left part of Fig. 3. The content of ARINC
653 standard is divided into five parts: overview, system func-
tionality, service requirements, configuration, and verification.
Our work formalizes the system functionality (including health
monitoring) and service requirements, which are the main parts
of the standard. Chapters 1, 2.1, 2.2 and 3.1 give an overview
of ARINC 653 from different perspectives, which are a high level
description for easy understanding and therefore need not to be
modelled. The configurations described in 2.5 and 5 define the
information and data format of the system configuration for
integration and deployment, and are eliminated in our model.
Notice that the chapter of memory management in ARINC 653
does not define any service, and therefore is also eliminated in
our model. Also, we omit other sections not mentioned here
and not providing enough details/information to construct a
formal model of them.

The second aspect into consideration is how to model. The
system functionality is informally defined in natural language
and can only be modelled manually. From this informal
description, we extract components and their attributes, actions
on components and their effects, and constraints. These elements
are all represented in Event-B as sets, constants, variables,
events, and invariants. Service requirements are presented by
the APEX service specification grammar, which is a structured language. It provides the possibility of semi-automatically translating these requirements to Event-B model. We design an algorithm to guide manual translation which is discussed in the next section.

Model structure. According to the document structure, we design our model formalization as shown in the right part of Fig. 3. We firstly formalize the system functionality of partition, process, and time management. The Event-B machine Mach_Part_Trans models the partition operating modes. Mach_PartProc_Trans refines the partition operating modes, and adds process state transitions. Mach_PartProc_Trans_withEvents defines all events of partition, process, and time management according to the system functionality. Mach_PartProc_Manager formalizes the service requirements of the partition, process, and time management. Then, we add the system functionality and service requirements of the communication: Mach_IPC_Conds specifies the functionality and events of inter-partition and intra-partition communication, and Mach_IPC formalizes their service requirement. Finally, the system functionality and service requirements of the health monitor are formalized in Mach_HM.

Verification. The main verification approach in Event-B is deductive reasoning of proof obligations. Invariants should be preserved on machines and refinement between a refined machine and its refinement (we mainly use guard strengthening and simulation). According to the model structure, the consistency of the system functionality and the service requirements is proven by refinements between Mach_PartProc_Trans_withEvents and Mach_PartProc_Manager, and Mach_IPC_Conds and Mach_IPC. ARINC 653 defines the safety functionalities and variable’s data type. Beside that, it does not explicitly define any property. The data type of each variable is defined as an invariant on each variable in Event-B machines. We have also extracted all possible properties from the ARINC 653 standard, which are safety properties [37] as shown in Table I. As explained in Section V, other extractable properties, such as liveness, are not covered.

From the informal description of ARINC 653, we use manual modelling of the system functionality and manual translation of the service requirements. Hence, we have to manually validate the errors found in our Event-B formalization. After finding an error in the Event-B model, we check corresponding description in the ARINC 653 standard to confirm and locate the error.

IV. FORMALIZING ARINC 653

This section presents a formalization of ARINC 653 in Event-B. We first discuss our formalization criteria, then the Event-B model of the key system functionalities\(^1\), and finally how service requirements are formalized by translating the service specification grammar into Event-B.

A. Formalization criteria

The criteria of modeling Event-B based systems are how to represent the system components, their attributes, safety prop-

---

\(^1\) The complete Event-B model can be downloaded from GitHub, [http://github.com/ywzh/arinc653model](http://github.com/ywzh/arinc653model)
Components and types. The main components in ARINC 653 are: partition, process, communicating components (port, channel, buffer, blackboard, semaphore, and event), waiting queue, and error handler. These components can be classified as statically configured components and dynamic components. Partitions are static components configured at build time and initialized during the POS booting. Any other component is a dynamic component, which is only created during partition initialization, using services ARINC 653 CREATE *. In particular, ARINC 653 does not allow creating components during partition runtime. We use sets in Event-B to represent partitions. Process and communicating components are represented by sets and variables. Sets specify the valid domain of a component, whilst variables are used to keep track of the created components during initialization. For instance, set PROCESSES defines the domain of all processes in a system and variable processes stores already created processes, and is a subset of PROCESSES. Waiting queues and error handlers are associated with communicating components and partitions respectively, and are represented as variables.

Component attributes. ARINC 653 defines the attributes of each component including fixed attributes and variable attributes. Fixed attributes can not be changed during runtime and are defined as constants in Event-B, and variable attributes are defined as variables. These constants and variables are functions mapping from a component’s set to the data type of the attribute. Event-B supports functions with different properties, such as partial functions, total functions, and partial injections. For instance, the Period of Partition constant is a partition attribute, and represents a total function Period of Partition ∈ PARTITIONS → ℤ, to indicate that all partitions have a period attribute of type natural. Whilst, the deadline time of process process attribute is a partial function deadline time of process ∈ processes → ℤ, since deadline time is undefined when a process is stopped, and defined when the process is in any other state.

Component relations. Due to spatial separation of ARINC 653, each component is created in one partition and it is bound to that partition. These relations may be one-to-one or one-to-many, which are represented with different types of functions in Event-B. For instance, errorhandler of partition ∈ PARTITIONS → processes is a partial injection since each partition has at most one error handler process.

Control and actions. ARINC 653 defines control and actions on components, such as the partition control, the process control, and the port control. Each action on these components are formalized with one or more events in Event-B. Due to the semantic gap between sequential description of service requirements and the guard-action style event model in Event-B, we carefully considered the design principle of events and used semi-automatic translation from service requirements to Event-B as discussed in Subsection IV-C.

B. Event-B model of key system functionality

This subsection introduces the Event-B model for some of the key system functionalities in ARINC 653.

1) Partition and process: Partitions and processes functionality mainly considers partition control (operating modes and transition), process control (process states and transitions), and the scheduling principle. Since time management considers the timing aspect on control of process execution, we also model this functionality in this part.

Process state transitions are complex since process states are dependent on partition operating modes. Process states and transitions are encoded in Event-B in machine Mach_PartProc_Trans as it follows. From the guard of the process state transition event (grd20 and grd21), we can analyze the nesting between partition operating modes and process states. This event models most of the state transitions, however some state transitions are modelled separately in other events because they are a consequence of other systems events. For instance, transitioning from state Ready to state Running is triggered by the process scheduling, therefore this transition is modelled in event scheduling; the state transitions from process states in the COLD START or WARM START partition modes to states in the NORMAL mode are modelled in event partition modetransition to normal; the creation of a process is modelled in event create_process, hence the transition from state Dormant to Ready is carried out in that event. In ARINC 653, state Waiting implies three situations, a process was suspended, waiting for a resource, or suspended when waiting for a resource. To explicitly distinguish these situations, we define three states to represent them Suspended, Waiting and WaitingSuspend respectively.

Process state transitions in Event-B only model the possible transition path, not the actions executed during the transitions. These are modelled in machine Mach_PartProc_Trans with Events, which models the process control, and defines the actions triggered by state transitions by means of the events suspend, suspend_self, resume, stop, stop_self, start, delayed_start, timed_wait, period_wait, time_out, req_busy_resource, resource_become_available, among others, for the concrete process control. In these events, we strengthen the guards of process state transition and those events defined in machine Mach_PartProc_Trans with Events, in such a way that it is refined by Mach_PartProc_Trans with Events.

2) Timing and scheduling: Timing and scheduling functionalities are important features of safety-critical real-time systems. Timing is not explicitly specified in ARINC 653 as a system functionality, except time management for processes.
We provide a simple model of timing and a two-level scheduling, to schedule IMA partitions and processes.

**Scheduling.** In POSs, all actions are initiated whenever a significant event occurs but scheduling, which is initiated by the regular event of a clock tick. The tick-tock has been considered as a way to model discrete time in Event-B [38]. We use event *ticktock* to represent the regular event of a clock tick, which increases variable *clock_tick* in one unit and variable *need_reschedule* is set to *TRUE* when appropriate to trigger the scheduling. The scheduling specified in ARINC 653 is a two-level scheduling. Partition scheduling is a fixed, cycle based scheduling and is strictly deterministic over time. The cyclic scheduling consists of a major time frame (MTF) that is split into partition time windows (PTW), each PTW having starting and ending time relatives to the starting time of the MTF. Each PTW of a MTF is associated to a given partition, which is executed when the system time reaches the starting point of the PTW and a new partition scheduling is performed when the system time reaches the end of the PTW. Process scheduling is priority preemptive and carried out by the partition.

The scheduling process chooses the current partition and process under execution according to the current PTW and the process priority. The partition scheduling is enabled when variable *need_reschedule* is *TRUE*, and the current time is in a PTW different than the current one. Note that the IDLE partition can not be scheduled. Variable *need_reschedule* is enabled by event *ticktock* or by the events in which the running process is blocked. We also define variable *need_procesch* to indicate process scheduling after partition scheduling. If the partition mode is NORMAL, an ARINC process should be chosen to be executed. Otherwise, the partition is at the START mode and the main process of this partition occupies the processor time.

There are two special types of ARINC processes that need to be carefully considered during process scheduling: the error handler and processes locking the process preemption of its partition. A process can lock the process preemption of its partition to prevent process rescheduling when accessing a critical section or a resource shared by multiple processes of the same partition. The error handler is a special aperiodic process with the highest priority, without deadline, and is invoked by the POS when a process level error is detected. It preempts any running process regardless of its priority and even if preemption is locked.

The abstract meaning of process scheduling for a partition in the NORMAL mode is to choose a process to run. The current running process is placed into the Ready state and the chosen process into the Running state. If the chosen process is the current running process, it remains in the Running state. The abstract event of process scheduling is shown as follows.

The *process_schedule* abstract event is extended in machine *Mach_PartProc_Manager* by events *process_schedule* and *run_errorhandler_preeptor*. The first event represents the normal process scheduling in a partition, whilst the second one represents the situation where the error handler of the partition has been started or a process has locked the preemption process of this partition.
Fig. 5. The message based interpartition communication (queuing mode)

out, the POS initiates a time counter for that time-out. After the time elapses, the POS responds to the expiration of the time-out by starting the process or sending a time-out message.

We define variable \( \text{timeout\_trigger} \in \text{processes} \rightarrow (\text{PROCESS\_STATES} \times \mathbb{N}_1) \) to store the time counter for processes. \((\text{proc} \mapsto (\text{PS\_Ready} \mapsto t)) \in \text{timeout\_trigger}\) means that a process \(\text{proc}\) is blocked waiting for some resource until time \(t\), moment at which the blocked process is placed into state \(\text{Ready}\). In the model, a tuple \((\text{ps} \times t)\) is inserted into \(\text{timeout\_trigger}\) when the POS initiates a time counter with a duration \(t\), and it blocks the process. Then when \(t\) arrives, the event \(\text{time\_out}\) triggers a blocked process to state \(\text{ps}\).

3) Interpartition communication: Interpartition communication is conducted via messages in ARINC 653. A partition is allowed to exchange messages through multiple channels via their respective source and destination ports. Ports can be configured into two different modes which determine the communication mode: queuing and sampling modes. Interpartition communication related actions and events are modelled in the machine \(\text{MACH\_IPC\_Conds}\).

Messages are atomic entities in ARINC 653. For this reason, we define set \(\text{MESSAGES}\) to represent all abstract messages in the system, and variable \(\text{used\_messages}\) to represent the set of already sent messages (used). In interpartition communication sending services, messages to be sent should be in set \(\text{MESSAGES}\) \(\setminus\) \(\text{used\_messages}\), and stored into \(\text{used\_messages}\) after being sent.

Each port has a message space to store the message(s) to be sent/received via this port. On the one hand, sampling ports have a single message storage, which is modelled using variable \(\text{msgspace\_of\_samplingports} \in \text{SamplingPorts} \rightarrow (\text{MESSAGES} \times \mathbb{N}_1)\). On the other, queuing ports keep a message queue, which is represented by variable \(\text{queue\_of\_queuingports} \in \text{QueuingPorts} \rightarrow \mathcal{P}(\text{MESSAGES} \times \mathbb{N}_1)\). A pair in \(\text{MESSAGES} \times \mathbb{N}_1\) means a message sent/received at a specific time. For instance, if the service of writing sampling message \(m\) has been invoked on port \(p\) at time \(t\), then the ordered pair \((p \mapsto (m \mapsto t)) \in \text{msgspace\_of\_samplingports}\).

“The communication between partitions is done by processes which are sending or receiving messages. In queuing mode, processes may wait on a full message queue (sending direction) or on an empty message queue (receiving direction). Processes waiting for a port in queuing mode are queued in FIFO or priority order.” [2] We use variable \(\text{processes\_waiting\_for\_queuingports} \in (\text{processes} \times \mathbb{N}_1 \times \text{MESSAGES}) \rightarrow \text{QueuingPorts}\) to store waiting processes of a queuing port. An ordered pair \((\text{proc} \mapsto (t \mapsto m)) \mapsto \text{port} \in \text{processes\_waiting\_for\_queuingports}\) means that process \(\text{proc}\) invoked the service of sending/receiving a message \(m\) to/from \(\text{port}\) on the time \(t\) and was blocked. ARINC 653 specifies two different policies to unblock processes blocked in a queuing port: based on the highest waiting time, where the unblocked process is that one which has been blocked in the queue for a longer time, or based on the process priority, where the unblocked process is that one with a higher priority among those blocked in the queuing port. This policy is statically specified for each queuing port. The message based communication in the queuing mode among partitions is shown in Fig. 5.

The port control system functionality in interpartition communication describes the communicating actions including creating ports, and sending/receiving message(s) to/from a sampling or queuing port. These actions are specified as events indicating the guards and the result of the port control action, which is proven to be strengthened and simulated by machine \(\text{MACH\_IPC}\), which encodes the service requirements, hence refining machine \(\text{MACH\_IPC\_Conds}\). The event \(\text{send\_queuing\_message}\) is modelled as it follows.

```
send_queuing_message \in \text{any\_port}\ x\ \text{msg} \text{where}
@grd01\ x \in \text{PARTITIONS};
@grd02\ x \in \text{ports};
@grd03\ \text{Direction\_of\_Ports}(x) = \text{PORT\_SOURCE};
@grd04\ \text{msg} \in \text{MESSAGES} \land \text{msg} \notin \text{used\_messages};
\exist\ t (t \in \mathbb{N} \land (\text{msg} \mapsto t) \in \text{queue\_of\_queuingports}(x)) <
\exist\ N (\text{Max\_Msg\_Num\_of\_QueuingPorts}(x) < N); \text{processes\_waiting\_for\_queuingports}(x) - [\{x\}] = \emptyset;
\then\ x \in \text{queue\_of\_queuingports};
\exist\ t (t \in \mathbb{N} \land (\text{msg} \mapsto t) \in \text{queue\_of\_queuingports}(x));
@act01\ \text{used\_messages} := \text{used\_messages} \cup \{\text{msg}\};
@act02\ \text{used\_messages} := \text{used\_messages} \cup \{\text{msg}\};
```

This event encodes the action of sending a message \(\text{msg}\) via a queuing port \(\text{port}\) when the port is not full and there is no other process waiting to send a message through this port. Guard \(\text{grd05}\) ensures that the current number of messages of the queue is less than the queue size. Guard \(\text{grd06}\) ensures that there is not any other process waiting for sending messages via this port. Otherwise, the process invoking the event needs to be blocked. The actions state that message \(\text{msg}\) is in the queue of \(\text{port}\) (\(\text{act01}\)) and has been used (\(\text{act02}\)) after the execution of this event.

4) Intrapartition communication: Intrapartition communication mechanisms are buffers, blackboards, semaphores, and events. For space reasons we only introduce the blackboard model.

A blackboard allows to send/receive messages to/from processes belonging to the same partition. Any message written on it remains there until the message is either cleared or overwritten by a new instance of the message. When a process attempts to read a message from an empty blackboard, it is queued for a specified amount of time. When a message is displayed on the blackboard, the POS removes from the process queue all the processes waiting for that blackboard and puts them in state \(\text{Ready}\).
Variables $\text{msgspace_of_blackboards} \in \text{blackboards} \rightarrow \text{MESSAGES}$ and $\text{emptyindicator_of_blackboards} \in \text{blackboards} \rightarrow \text{BLACKBOARD_INDICATORTYPE}$ store the message that has been displayed on a blackboard and respectively indicate whether a blackboard is empty or not. If $\text{emptyindicator_of_blackboards}$ is not false then $\text{msgspace_of_blackboards}$ is different from the empty set, therefore indicating that the blackboard contains at least one message. Events $\text{display_blackboard}$ and $\text{display_blackboard_needwakeuprdprocs}$ encode the functionality of displaying a message in the blackboard when there are not any waiting process for that blackboard and when there are any waiting process that have to be awakened. Similarly, events $\text{read_blackboard}$ and $\text{read_blackboard_whenempty}$ respectively encode the reading functionality when the blackboard is not empty or it is empty and the process has to be enqueued in the waiting process queue. Event $\text{display_blackboard_needwakeuprdprocs}$ extends the event $\text{resourceBecomeAvailable2}$ that wakes up all processes waiting for a resource, and is enabled when the waiting process queue is not empty (grd505). Actions act501 $\sim$ act504 represent the result of this event. The Event-B model is shown below.

\[
\text{display_blackboard_needwakeuprdprocs} \text{ extends resourceBecomeAvailable2} \equiv \\
\text{any } \text{bb msg where} \\
\text{grd506 } \text{bb} \in \text{blackboards}; \\
\text{grd506 } \text{msg} \in \text{MESSAGES} \land \text{msg} \notin \text{used_messages}; \\
\text{grd506 } \text{processes_waiting_for_blackboards} \neg \{(\text{bb})\} \neq \emptyset; \\
\text{then} \\
\text{act501 } \text{msgspace_of_blackboards}(\text{bb}) := \text{msg}; \\
\text{act502 } \text{processes_waiting_for_blackboards} := \text{procs} \cup \\
\text{act503 } \text{used_messages} := \text{used_messages} \cup \{\text{msg}\}; \\
\text{act504 } \text{BB_OCCUPIED}; \\
\text{end}
\]

5) **Health monitoring:** In ARINC 653, HM is responsible for responding to and reporting hardware, application and POS software faults and failures. ARINC 653 supports HM by providing HM configuration tables and an application level error handler process. These tables are the Module HM table, the Multi-Partition HM tables, and the Partition HM tables. The error handler is a special process of the partition with the highest priority and no process identifier.

The HM is modelled in machine Mach_HM. It defines variable $\text{module_shutdown} \in \text{BOOLE}$ to control the module, and each event in machine Mach_IPC are extended in Mach_HM by adding a guard $\text{module_shutdown} = \text{FALSE}$. This means that if the module is shut down, no event can be triggered.

The HM decision logic is implemented in the guards of the events encoding recovery actions. Recovery actions at the module level are only triggered when the error is inside a partition time window (PTW) and the error is a module level error of the partition executed during the current PTW. Recovery actions at the partition level are triggered when the error is a partition level error and the error handler has not been created in that partition, or the error was caused by the error handler of the partition. For instance, the event $\text{hm_recoveryaction_shutdown_module}$ is specified as it follows.

```
hm_recoveryaction_shutdown_module \equiv \text{any errcode where} \\
\text{grd706 module_shutdown} = \text{FALSE}; \\
\text{grd701 errcode} \in \text{SYSTEM_ERRORS}; \\
\text{grd702 errcode} \in \text{dom(MultiPart_HM_Table(part))}; \\
\text{errcode} \rightarrow \text{MLA_SHUTDOWN} \in \\
\text{MultiPart_HM_Table(part)}; \\
\text{then} \\
\text{act701 module_shutdown} := \text{TRUE}; \\
\text{end}
```

Finally, if the error level is process, the error handler of this partition has been created, and the error was not caused by the error handler, the error handler is activated to deal with this error. How to handle an error is application dependent. Event $\text{hm_recoveryaction_errorhandler}$ is specified as it follows.

```
hm_recoveryaction_errorhandler \text{ extends start_periodic_process_innormal} \equiv \text{any errcode where} \\
\text{grd708 module_shutdown} = \text{FALSE}; \\
\text{grd701 errcode} \in \text{SYSTEM_ERRORS}; \\
\exists \text{errcode} \in \text{domain(MultiPart_HM_Table(part))) \land} \\
\text{ERROR_LEVEL_PROCESS} \rightarrow \text{errcode} \in \\
\text{dom(MultiPart_HM_Table(part)(errcode)));} \\
\text{DEADLINE_MISSED} \in \text{ran(MultiPart_HM_Table(part)(errcode))} \rightarrow \text{clock_tick > ONE_TICK_TIME >} \\
\text{deadlinetime_of_process(pro))}; \\
\text{grd704 part} \in \text{dom(errorhandler_of_partition);} \\
\text{grd705 current_process} \neq \text{errorhandler_of_partition(part);} \\
\text{grd706 proc} = \text{errorhandler_of_partition(part);} \\
\text{end}
```

C. Translating service requirements into Event-B

A service definition in the APEX specification contains the name of the service and a list of formal parameters ($p_1 \ldots p_n$) with their types. The description of a service consists of a short definition of its functionality and a full description of its semantics. The semantic description gives the algorithm of the service behavior, which is composed of two parts: an error part which describes error handling due to incorrect values of actual input or input-output parameters, and a normal part which describes the treatment to be performed when no error is detected by the service.

Although ARINC 653 defines a structured language to describe the functional requirements of APEX services, we find that ARINC 653 Part 1 only uses compound statements “IF” and “SEQUENCE” to describe complex structures. Moreover, due to the natural language description of simple statements, it is difficult to implement automatic translation from APEX specification to Event-B model. Hence, we concentrate on translating the structure of APEX services to events, and the detailed behavior of each service represented by simple statements needs to be modelled by hand. The specification grammar for APEX services is illustrated in the left part of Fig. 6, and the generated events are shown in the right part.

Event-B does not have compound statements such as “IF”, “CASE”, and “LOOP”, therefore the description of an APEX service should be decomposed into events with non-intersecting guards. That is, if one event is enabled by its guard, then all of other events are disabled. For the “IF” statement, its body (e.g., acts1, acts2, acts3) are behaviors under different conditions that do not intersect. Therefore, we use an event to represent the behavior of each body. Additionally, for each “IF”...
Fig. 6. Translate APEX specification into Event-B

statement we need to add a new event representing the “ELSE” body, even when it is and “IF” without an “ELSE”. The type of
parameters of the APEX service is encoded as guards of each event \((p_1..._type)\). In the error part, conditions \(gcond_1, gcond_2\), etc. indicate incorrect values of parameters, and thus their negation are translated into Event-B as guards of each event. The conditions of the “IF” statement are also translated as guards. In Fig 6, the condition of \(act_{s11}\) is \(cond_1 \land cond_{11}\), so the guard of the corresponding event includes \(cond_1 \land cond_{11}\). Whilst, the condition of \(act_{s12}\) is \(cond_1 \land \neg cond_{11} \land cond_{12}\).

A simple statement, such as “set the specified process to state DORMANT,” in Fig. 2, is translated into actions in the event according to the meaning of the statement, which is usually represented by a deterministic assignment, e.g.,

\[
\text{process\_state}(\text{PROC}) := PS\_DORMANT.
\]

We have designed an algorithm to guide manual translations from APEX service requirements into Event-B models as shown in Algorithm 1. For convenience, we first give a simple syntax for the APEX service specification grammar:

\[
c ::= \text{ACT } \text{act} \mid c_1; c_2 \mid \text{IF } \text{cond} \text{ THEN } c_1 \mid \text{ELSE } c_2
\]

where \text{ACT } \text{act} is a simple statement and \(c_1; c_2\) is the sequence statement.

The algorithm translates a service requirement presented in this syntax into a set of events. A service requirement is a tuple \(<\zeta, P, E, S, >\), where \(\zeta\) is the service name, \(P\) is a parameter list, \(E\) is the error conditions in the error part, and \(S\) is the normal part. In Algorithm 1, function \text{translate} translates a statement into a set of events and \text{translate\_service} translates a service requirement into the final set of events. An event in the \(evts\) set is a tuple \((\sigma, \alpha, \sigma')\), where \(i\) is name of the event, \(\sigma\) is a list of guards, and \(\alpha\) is the list of actions of the event. We initially put an event with empty guards and empty actions into set \(evts\). The actions for simple statements is immediate, but when it is and “IF” without an “ELSE”. The type of

\[
\text{Algorithm 1: Translate APEX service into Event-B}
\]

\[
\text{function translate(evts,stmt)}\begin{cases}
\text{switch stmt do}
\begin{align*}
\text{case ACT } & \text{act} \quad \text{add the action act to end of action list of each event in evts; return evts;}
\text{case stmt1;stmt2} \quad \text{evts' } \leftarrow \text{translate(evts, stmt1); return translate(evts', stmt2);}
\text{case IF } \text{cond} \text{ THEN } \text{stmt1 ELSE stmt2} \quad \text{evts' } \leftarrow \text{translate(evts, stmt1); return translate(evts', stmt2);}
\text{case IF } \text{cond} \text{ THEN stmt end ;stmt end } \quad \text{evts' } \leftarrow \text{duplicate of evts; add the "cond" to end of guard list of each event in evts; return evts ∪ evts'}
\text{case IF } \text{cond} \text{ THEN st end } \quad \text{evts' } \leftarrow \text{translate(evts, stmt); return evts ∪ evts'}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\text{function translate\_service(spec)}\begin{cases}
\text{switch spec do}
\begin{align*}
\text{case <ζ,P,E,S>} \quad \text{evts } \leftarrow \{<ζ, α, φ >\}; \text{ evts } \leftarrow \text{translate(evts, S)} \quad \text{add } \wedge (\neg E_i) \text{ to guard list of each event in evts return } \{\text{ev, ev ∈ evts }\mid \text{ev.α } ≠ \emptyset\}
\end{align*}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{current\_process\_flag } & = \text{TRUE} \land \\
\text{part } & \in \text{dom(error\_handler\_of\_partition)} \\
\text{@grd45 current\_process } & = \text{error\_handler\_of\_partition(part)} \land \\
\text{proc } & = \text{process\_call\_error\_handler(current\_process)} \\
& \Rightarrow \text{newlocklevel } = \text{[part } \rightarrow 0\}; \\
& \neg \text{current\_process\_flag } = \text{TRUE} \land \\
\text{@grd46 current\_process } & = \text{error\_handler\_of\_partition(part)} \land \\
\text{proc } & = \text{process\_call\_error\_handler(current\_process)} \\
& \Rightarrow \text{newlocklevel } = 0; \\
\end{align*}
\]

\text{grd45} means that if the current process is the error handler of the current partition and the process to be stopped is the process which the error handler preempted, then newlocklevel is an ordered pair \(\text{part } \mapsto 0\). Otherwise, newlocklevel is \(\emptyset\). In the actions of event \text{stop}, there is an assignment locklevel\_of\_partition := locklevel\_of\_partition \leftarrow
newlocklevel. Thus, the semantic of the "IF" statement is implemented without decomposing into different events.

V. RESULTS AND DISCUSSION

A. Model and proof statistics

In Table II we give model and proof statistics of ARINC 653 Part 1 in the Rodin tool. These statistics measure the size of the model, the proof obligations generated and discharged by the Rodin tool, and those proved interactively. The LOC of the machines increase gradually since the refinement machine is an extension of the refined machine. Therefore, the total LOC is not the summation of all machines and is not counted here.

Proving is a time-consuming and skilled work in Event-B. Fortunately, the Rodin tool provides effective automatic provers that saves much proving time on our model. In addition it is possible to integrate third-party provers (such as Atelier B prover) and SMT solvers (such as CVC3 and Z3) as Rodin plugings bringing the degree of automation to a higher level (more than 80%).

B. Errors found in ARINC 653

We found three errors in ARINC 653 Part 1, one in the process management service, and two in the inter- and intra-partition communication services. Additionally we detected three cases where the specification of process state transitions is incomplete.

1) In process state transitions: An incomplete description of process state transitions is detected. The "process states and state transitions" description in ARINC 653 Part 1 is very detailed and try to list all actions triggering transitions. But we find that some significant actions are not involved either in the figure or in the text.

The errors are the following:

- in the COLD/WARM_START mode, a suspended process can also be resumed and its state transits from Waiting to Waiting. This action is missed in the "Waiting - Waiting" transition conditions in the standard.
- in the NORMAL mode, if an aperiodic process is delayed_started (i.e., the delay time > 0), its state transits from Dormant to Ready. This action is missed in the "Dormant - Waiting" transition conditions in the NORMAL mode in the standard.

These incompleteness are found by verifying the guard strengthening between machine Mach_PartProc_ Trans_withEvents and its refinement machine Mach_PartProc_Manage. The first machine models the process state transitions and their triggering actions according to the standard, and the refinement models the service requirements. The guard strengthening requires that the state transitions specified in the refinement should not be contradictory with the transitions in Mach_PartProc_Trans_withEvents. After carefully checking these errors, we find that the service requirements are correct while the process state transitions are incomplete.

2) In requirement of process management services: The error is in the requirements of the RESUME service. In fact, an aperiodic process that has been delayed_started is in the Waiting state if it is suspended. When that process is resumed, it should retain in the Waiting state if the delay time has not been reached. But according to service requirement RESUME defined in ARINC 653 partly shown below, the aperiodic process is set into the Ready state, because this process is not waiting on a process queue or on a TIMED_WAIT time delay. This error is found by verifying the guard strengthening between the resume events in the machine Mach_PartProc_Trans_with_Events and Mach_PartProc_Manage.

3) In requirements of the communication service: We find two errors in the requirements of the communication service.

The first one is an error in inter-partition communication. The SEND_QUEUEING_MESSAGE service is used to send a message via a specified queuing port. If there is enough space in the queuing port to accept the message, the message is inserted to the end of the port’s message queue. If there is not, then the process is blocked and stays in the waiting queue until the specified time-out, if finite, expires, or space becomes free in the port to accept the message. However, in the service specification when the time-out does not expire and space is released, the sent message is not inserted into the message queue. This leads to an error where sending processes waiting for the free space within the time-out loses their sent messages. This error is found by verifying the simulation between the send_queueing_message events in the machine Mach_PartProc_Trans_with_Events and Mach_PartProc_Manage. The event in the first machine requires that the message should be in the message queue after being sent when the message queue is sufficient, or not sufficient but the sending process does not expires. But the second machine is contradictory with it.

<table>
<thead>
<tr>
<th>Machine</th>
<th>LOC</th>
<th>Proof obligations</th>
<th>Automatically discharged</th>
<th>Interactively discharged</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach_Part_Trans</td>
<td>39</td>
<td>7</td>
<td>6 (86%)</td>
<td>1(14%)</td>
</tr>
<tr>
<td>Mach_PartProc_Trans</td>
<td>236</td>
<td>128</td>
<td>118 (92%)</td>
<td>10(8%)</td>
</tr>
<tr>
<td>Mach_PartProc_Trans_withEvents</td>
<td>495</td>
<td>223</td>
<td>219 (98%)</td>
<td>4(2%)</td>
</tr>
<tr>
<td>Mach_PartProc_Manage</td>
<td>900</td>
<td>580</td>
<td>504 (87%)</td>
<td>76(13%)</td>
</tr>
<tr>
<td>Mach_IPC_Conds</td>
<td>2022</td>
<td>272</td>
<td>169 (62%)</td>
<td>103(38%)</td>
</tr>
<tr>
<td>Mach_Instance</td>
<td>2267</td>
<td>560</td>
<td>463 (83%)</td>
<td>97(17%)</td>
</tr>
<tr>
<td>Mach_HIM</td>
<td>2712</td>
<td>18</td>
<td>9 (50%)</td>
<td>9(50%)</td>
</tr>
<tr>
<td>Total</td>
<td>1791</td>
<td>1491(83%)</td>
<td>300(17%)</td>
<td></td>
</tr>
</tbody>
</table>
The second error is in the specification of the RECEIVE_BUFFER service in intra-partition communication. This service is used to receive a message from a specified buffer. When the buffer is not empty, the receiving process can receive a message directly, and the message should be removed from the message queue of this buffer. But in the service specification, we find that the received message is not removed. This leads to an error that the message queue of a buffer may always be full. This error is found by verifying the simulation between the receive_message events in machines Mach_PartProc_Trans_with_Events and Mach_PartProc_Manage. The event in the first machine requires that the message should not be in the message queue after being received, but the second machine is contradictory with it.

C. Discussion

1) Completeness of our model: Although we have formalized all of ARINC 653 services and the system functionality, some implementation-related details are not covered in our formalization: (1) we eliminate execution context attributes of partitions and processes, e.g. entry point and stack size, since ARINC 653 only defines these attributes, but do not provide any functionality for them. (2) we omit the detailed partition switch, that leads us to capture only the error inside a partition time window in the HM. (3) we do not model the partition switching, that leads us to forgo the verification result. However, these details should be implemented when formally developing a POS from the ARINC 653 specification.

2) Event-B modeling of software specification: The most notable problem when we are formalizing the ARINC 653 is the semantic gap between sequential description of the service requirements and guard-action style event model. ARINC 653 specifies the software, i.e. the partitioning operating system, therefore the semantics of each service are specified by a structural sequential language, whilst Event-B is a formalism for developing and verifying systems using events. Sequential programs can be generated from the Event-B model, while to our knowledge, there is no known work on how to translate structural languages into Event-B. This paper provides a preliminary translation approach from APEX specification grammar to Event-B, but an ideal and semantically equivalent translation framework is needed for high assurance.

3) Deductive verification vs. model checking: Model checking is an automatic and “push-down” verification approach for system and software. As concluded in [12], verification of complex systems is never automatic or “push-button”. This standpoint is also confirmed in our work. In the Rodin tool, the ProB [39] is a model checker for the Event-B models. At first, we try to furnish properties in linear temporal logic and model check them using ProB. We find that it is feasible on our model on the first and second levels of abstraction, but the state space is too large to be checked on other levels. The deductive verification in Event-B is done by logic reasoning. Although it needs expert skills in mathematics and logics, most of proof obligations can be discharged automatically by provers (up to 80%).

4) Safety and liveness properties: The shortcoming of deductive verification in Event-B is that properties are represented as invariants, which are safety properties. Liveness properties are another major class of properties [37]. In [40], the authors collected 555 examples of property specifications, 27.2% are invariant properties (global absence and universality properties), whilst, liveness properties cover about 45.6% (global existence and response). We found that liveness properties are very useful in the specification of the communication in ARINC 653. Liveness properties can partially be expressed by the Flow language in [41], but it is difficult to specify possible continuation scenarios, which means that globally (G) quantifier is not supported. Reasoning about liveness properties directly on Event-B model is feasible [42], the absence of supporting tools makes it difficult to be applied to complex models that lead us to forgo the verification.

5) Reusing models for development and verification: The Event-B model of ARINC 653 defines the abstract specification of POSs. This model can also be used in the model-based development and the verification of POSs.

A POS is an execution environment for applications. Composition of the ARINC 653 and application models enables the simulation, analysis, and verification for IMA systems. This requires that the application is also modelled in Event-B and that tools in Event-B support complex analysis on the model. Current version of the Rodin tool only supports functional verification and simulation. In particular, time analysis needs to be strengthened in Rodin or else it is necessary to export the Event-B model to another analysis tools.

Second, the ARINC 653 model provides the possibility of formally developing a new POS by refinement, and finally generating the source code. The Rodin tool provides many code generation plugins, such as C, C++, Java and Ada. The difficulty of this goal may come from the hardware dependency of POSs and the efficiency of the generated code. The source code of POSs usually has some intrinsic patterns and contains assembly code. These require that the Event-B model is detailed enough and that the code generation in Rodin is revised according to this target.

VI. CONCLUSIONS

In this work, we have presented the formalization and deductive verification of the ARINC 653 standard using Event-B. The system functionality and all of 57 services specified in ARINC 653 Part 1 have been modelled in Event-B. The safety properties of the system functionality and service requirements, and the consistency between them are checked by discharging proof obligations of invariants and refinement preservation. Finally, we found three errors in ARINC 653 Part 1 and detected three cases where the specification is incomplete. The verification was significantly simplified due to the high degree of automated reasoning in the Rodin tool.

As future work we consider to include mechanical checking of these errors in ARINC 653 compliant OSs source code such as XtratuM, and POK, and to extract liveness properties from the ARINC 653 standard and discover appropriate solutions to verify them. Since ARINC 653 is being considered to support multicore platform, it is also an important aspect in our future work.
We much appreciate the suggestions from Prof. Jean-Paul Bodeveix and Prof. Mamoun Filali from IRIT, Université de Toulouse, France. This work is partially supported by the Fundamental Research Funds for the Central Universities in China (Grant No.YWF-15-GJSYS-083), and by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate.

REFERENCES


