Formal Specification and Analysis of Partitioning Operating Systems by Integrating Ontology and Refinement

Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu

***Abstract*—Partitioning operating systems (POSs) have been widely applied in safety-critical domains from aerospace to automotive. In order to improve the safety and the certification process of POSs, the ARINC 653 standard has been developed and complied with by the mainstream POSs. Rigorous formalization of ARINC 653 can reveal hidden errors in this standard and provide a necessary foundation for formal verification of POSs and ARINC 653 applications. For the purpose of reusability and efficiency, a novel methodology by integrating ontology and refinement is proposed to formally specify and analyze POSs in this paper. An ontology of POSs is developed as an intermediate model between informal descriptions of ARINC 653 and the formal specification in Event-B. A semiautomatic translation from the ontology and ARINC 653 into Event-B is implemented, which leads to a complete Event-B specification for ARINC 653 compliant POSs. During the formal analysis, six hidden errors in ARINC 653 have been discovered and fixed in the Event-B specification. We also validate the existence of these errors in two open-source POSs, i.e., XtratuM and POK. By introducing the ontology, the degree of automatic verification of the Event-B specification reaches a higher level.**

***Index Terms*—ARINC 653, Event-B, formal specification, ontology, OWL2, partitioning operating systems (POSs), refinement.**

I. Introduction

PARTITIONING operating systems (POSs) [1] are used to support applications shared access to critical resources within integrated systems. They provide an independent execution of applications by temporal and spatial partitioning. In order to improve the safety and the certification process of POSs, the ARINC 653 standard [1] has been developed to standardize the interface between POSs and application software as well as the system functionality of POSs. ARINC 653 is a premier standard of POSs and compliant POSs, such as VxWorks 653, INTEGRITY-178B, LynxOS-178, and PikeOS, have been widely applied in domains from aerospace to automotive. However, hidden inconsistencies or incorrectness in this standard could mislead system developers' understanding, causing failures or malfunctions in POSs, and hence a breakdown of applications. A rigorous and mechanically checked formalization of ARINC 653 helps to ensure the correctness of ARINC 653. Moreover, for the purpose of the formal development and verification of ARINC 653 compliant POSs and applications, formal specification covering most parts of the standard is highly desirable.

There are three challenges in formalizing and verifying real-world standards in the scope of POSs. First, due to a combination of natural languages and textual grammars used in the ARINC 653 standard, a gap between such informal requirements and formal methods should be addressed [2]. Second, reusability of the formal specification is important. The formal specification should not only be used for formal verification, but also to improve the automation of development, integration, and management of systems. Third, formalization and verification of operating systems (OSs) are usually manpower intensive, e.g., 20 person-years are invested in the seL4 project [3]. Especially, the ARINC 653 standard is highly complex, which contains more than 100 pages of informal descriptions. Therefore, the efficiency of the formalization and verification is critical.

In order to resolve these challenges, the concepts of ontology and refinement are integrated to formally specify and analyze POSs in this study. An ontology typically provides a common vocabulary to describe a domain of interest. It is a shared understanding of the domain among people, organizations, and systems to bridge the gap mentioned above. Due to the interoperability, an ontology can be used for the development, integration, and management of Integrated Modular Avionics (IMA) applications and thus improves the reusability of formal specifications. Refinement is the verifiable transformation of an abstract specification into a concrete specification and further into an implementation. The refinement of formal specifications eases the formalization of complex systems, provides the certifiable assurance and safety of POSs, and ensures the consistency between the design and its implementation by verified code generation. Finally, the integration of ontology and refinement can improve the efficiency of the formalization and verification via the automatic translation.

Manuscript received December 03, 2015; revised February 24, 2016, April 21, 2016, and April 22, 2016; accepted April 29, 2016. Date of publication May 17, 2016; date of current version August 04, 2016. This work was supported in part by the National Research Foundation, Prime Minister's Office, Singapore, under its National Cybersecurity Research and Development (R&D) Program and administered by the National Cybersecurity R&D Directorate under Award NRF2014NCRNCR001-30. Paper no. TII-15-1802.R3. *(Corresponding author: Y. Zhao.)*

Y. Zhao is with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China, and also with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: zhaoyw@buaa.edu.cn).

D. Sanán, F. Zhang, and Y. Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: sanan@ntu.edu.sg; fuzh@ntu.edu.sg; yangliu@ntu.edu.sg).

Digital Object Identifier 10.1109/TII.2016.2569414

This paper presents a novel methodology of formalizing an informally described standard of POSs—ARINC 653 Part 1 [1], by integrating ontology and refinement. Unlike existing works on formalizing POSs, ARINC 653, and even general-purpose OSs, a Web Ontology Language (OWL2) [4] based ontology of POSs (OWL-POS) is proposed as an intermediate model to formalize ARINC 653 using Event-B [5]. Techniques and tools with industrial maturity are adopted in this study. OWL2 is an ontology language for the Semantic Web with various concrete syntaxes that can be used to serialize and exchange ontologies. It has been applied in industry, such as control systems [6]. Event-B is a refinement-based formal method for system-level modeling and analysis. It has been applied in industry, such as satellite software [7] and medical systems [8]. In detail, the technical contributions of this study are as follows.[1]

1) An OWL-POS ontology is developed to formally represent concepts, relations, and constraints of POSs. OWL-POS is the first ontology in the domains of POSs and ARINC 653 in the literature.
2) An automatic translation from OWL-POS and an translation from service requirements in ARINC 653 into Event-B are proposed to alleviate enormous efforts needed in the formalization and formal analysis.
3) A complete Event-B specification for ARINC 653 compliant POSs is developed. The Event-B specification covers the system functionality and all 57 services specified in ARINC 653 Part 1. This Event-B specification is the most complete formal specification of ARINC 653 in the literature.
4) Ninety-five safety properties are specified as invariants in Event-B. During the formal analysis of the Event-B specification, six errors in ARINC 653 have been discovered and fixed in the Event-B specification. We also validate the existence of the errors in two major open-source and ARINC 653 compliant POSs, i.e., XtratuM [9] and POK [10].

The rest of this paper is organized as follows. In Section II, we introduce the background and related work. Section III presents the formalization and analysis methodology. The next three sections present OWL-POS, the translation into Event-B, and the Event-B specification, respectively. Then, Section VII presents the formalization and verification results along with some discussion. Finally, Section VIII gives the conclusion and future work.

II. Background and Related Work

A. POSs and ARINC 653

The ARINC 653 standard defines a partitioning architecture for safety-critical systems on a single-core computing platform. The major functionalities of POSs are partition/process management, time management, inter-/intrapartition communication, and health monitoring. The latest version of ARINC 653 published in 2010 is organized in six parts. Part 1 specifies the required services; hence, ARINC 653 compliant POSs are mandated to implement this part. Other parts are overview, extended services, conformity test, subset services, and required capabilities. Thus, this paper focuses on formalizing and verifying Part 1. Part 1 defines the system functionality of POSs using more than 40 pages of natural language descriptions. It also defines service requirements as APplication EXecutive (APEX) interface using more than 60 pages of descriptions in the APEX service specification grammar which is a combination of natural and structural languages. For instance, the STOP service of process management presented by the APEX grammar is illustrated in Fig. 1. In the service requirements, 57 required services are specified.

```
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
    set the specified process state to DORMANT;
    if (current process is error handler and PROCESS_ID is process which the error
        handler preempted) 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. 1. Requirements of the STOP service (from [1, p. 61]).

B. Preliminaries of OWL2 and Event-B

OWL2 has been standardized by the World Wide Web Consortium as an ontology language for the Semantic Web. OWL2 is capable of creating classes, properties, individuals and defining operations (e.g., union, intersection) to represent domain knowledge. It has been widely applied in a large set of research and application domains.

Event-B uses set theory as a modeling notation, uses refinement to represent systems at different abstraction levels, and uses mathematical proofs to verify consistencies between levels of refinements. In this paper, we choose Event-B as the formalism for POSs because of the following.

1) Event-B provides a well-supported language and an environment for system specification and refinements. Compared to other system modeling approaches, e.g., Petri Nets [11], the proof obligations of refinement relations and invariants can be automatically generated in its development environment—RODIN. The high degree of automatic verification in RODIN alleviates the manual efforts substantially. Moreover, it has been successfully applied in the industry [7], [8].
2) The inductive verification approach in Event-B avoids the state-space explosion problem of automatic approaches (e.g., model checking) when verifying complex POSs.
3) The concept of event in Event-B is suitable for modeling OSs, where hardware components, e.g., interrupters like clocks and timers, need to be well managed.

Event-B models are described in terms of Contexts and Machines. Each machine may reference (see) a context. Contexts

[1] All deliverables of this study including the ontology, the Event-B specification, and all proofs are available at "https://github.com/ywzh/"

specify the static part of a model and consist of sets, Constants, and Axioms. Machines specify the dynamic part and may contain a set of state, Variables; a conjoined list of predicates, Invariants, to constrain variables; and state transitions (called Events). Suppose a machine M, referencing a context with Sets s and Constants c, an event of M is represented as

$$E \;\widehat{=}\; \textbf{any } x \textbf{ where } G(s,c,v,x) \\ \textbf{then } v :\mid BA(s,c,v,x,v') \textbf{ end}. \qquad (1)$$

E is the event name, x is a list of parameters of the event, v is a list of variables of the machine, $G(s,c,v,x)$ are guards of the event that state the necessary condition for the event to occur, and $v :\mid BA(s,c,v,x,v')$ 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 of a model.

C. Related Work

Formal specification and verification of general-purpose OSs have been enforced in recent years [12]. A notable project of them is the formal machine-checked verification of the seL4 microkernel [3]. POSs are very different from general-purpose OSs, defining a temporal and spatial partitioning, two-level scheduling, and inter- and intrapartition communications. Formal specifications of POSs have been developed in industry and academia [13]–[15]. Formal verification has been used on POSs for safety/security certification in the industry, such as ARINC 653 compliant POSs (e.g., PikeOS [16], [17] and INTEGRITY-178B [18]) and hardware implementations of partitioning (e.g., the AAMP7G microprocessor [19]). Some formal models of POSs in these works are not based on ARINC 653. Others only cover a small part of functionalities and services in ARINC 653.

Due to the importance of POSs standard in safety-critical systems, formalization of ARINC 653 has been taken into account in the literature. Formal specification of the ARINC 653 architecture and components has been developed using `Circus` language [20], architecture analysis and design language [21]–[23], and PROMELA in the SPIN model checker [24]. Since the ARINC 653 services are not the emphasis of these works, only a small part of services are modeled.

In summary, research works in the literature do not provide a complete formal specification of ARINC 653 compliant POSs. Second, the existing formal specification and models of POSs are developed directly according to informal requirements or standards using formal methods. There is still a large space to improve the interoperability and reusability of formal specification for system integration and management. Third, formal verification performed so far usually time-consuming and need many manual efforts. This paper goes one step further by integrating ontology and refinement.

III. Methodology

This section introduces the proposed formalization and analysis methodology, which is shown in Fig. 2. ARINC 653 system functionality defines components and their attributes,

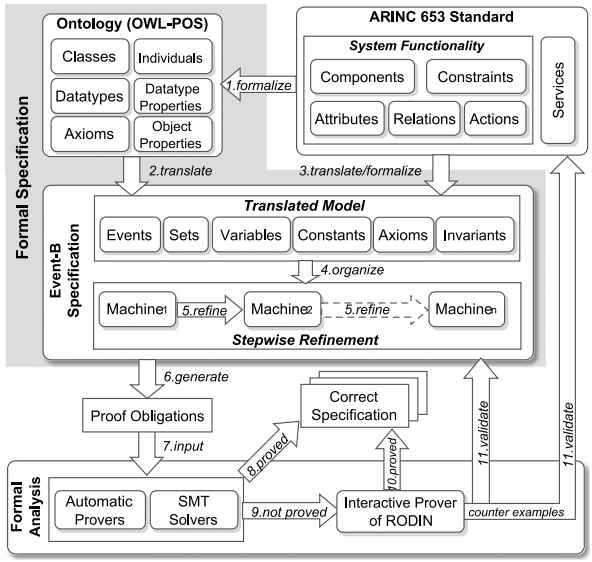

Fig. 2. Overview of methodology.

relations of components, control of components (actions), and constraints. The actions and services define the behavior of POSs, while the other elements define the structure of POSs. We formally model the structural part of ARINC 653 in OWL-POS (Step 1 in Fig. 2) using the OWL2 editor Protégé to create OWL-POS. Due to natural language descriptions of the system functionality, it is difficult to avoid manually developing the ontology representation of requirements. In order to reduce the effort of the formalization and analysis, automatic translation from OWL-POS into Event-B is enforced (Step 2). The behavioral part of ARINC 653 is semiautomatically modeled in Event-B (step 3). Actions in the system functionality are described in natural language and are manually formalized in Event-B. Service requirements are described in the APEX grammar, and we design an algorithm to guide their translation into Event-B. After these translations, we design a set of machines and contexts in Event-B as well as stepwise refinements between them to organize the generated models (Steps 4 and 5).

The verification objectives are to check: 1) whether the constraints are preserved on the system functionality and all the services defined in ARINC 653; and 2) whether the services correctly implement the system functionality. We use invariants and refinement verification supported in Event-B as the means for the two objectives, respectively. Proof obligations of invariants and refinement are automatically generated in RODIN (Step 6). In order to improve the degree of automatic verification, automatic provers (e.g., Atelier B prover) and SMT solvers (e.g., CVC3 and Z3) integrated in RODIN are first carried out on the proof obligations (Step 7). If the proof obligations cannot be automatically discharged, we try the interactive prover of RODIN (Step 9). When counterexamples are found, we manually validate the OWL-POS components and ARINC 653 to locate the errors and revise the formal specification to fix errors (Step 11).

In order to carry out the methodology, two challenges should be resolved. First, the mathematical foundation of OWL2 is description logic (DL), while Event-B are created on first-order

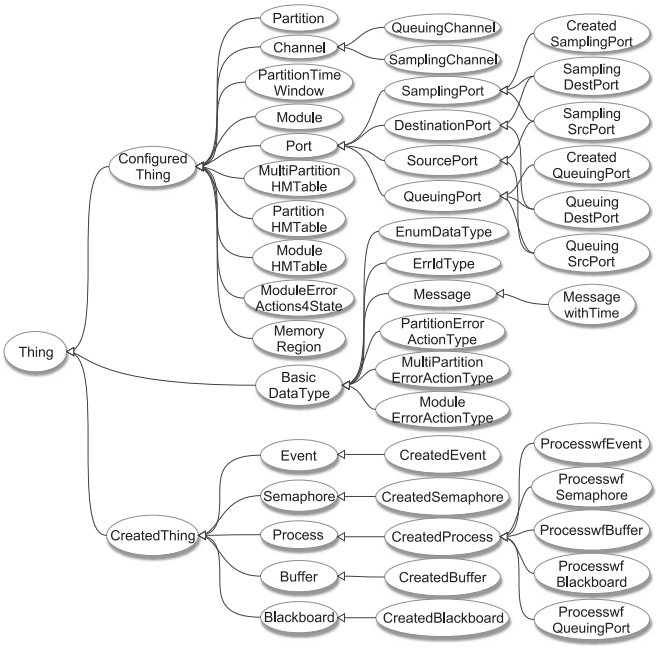

Fig. 3. Part of classes and inheritances in OWL-POS.

logic (FOL) and set theory. In order to translate OWL-POS into Event-B, the semantic gap between OWL2 and Event-B has to be bridged. Second, the APEX grammar is a combination of natural and structural languages, while an Event-B model is a discrete transition system, where an event is a transition and occurs when its guard is true. In order to translate the service requirements in the APEX grammar into Event-B, the semantic gap between them has to be bridged. In this paper, a semantic mapping from OWL2 to Event-B (OWL2EB) and an algorithm (APEX2EB) to guide translation from APEX grammar into Event-B are developed to resolve the two challenges, respectively.

IV. OWL-POS: An OWL2 Ontology of POSs

This section presents OWL-POS using the Manchester Syntax in OWL2.

A. Components and Attributes

The data structures of the POS components and their attributes are represented in the APEX grammar. All these components and attributes have been captured and formally represented in OWL-POS. Components are modeled as classes in OWL-POS. The data types of attributes in ARINC 653 are primitive data types (e.g., int, bool), enum data types, and composed data types. Attributes with primitive data types are modeled as data types. Attributes with enum data type are modeled as classes, which are subclasses of $EnumDataType$. Attributes with composed data types are modeled as classes, which are subclasses of $BasicDataType$. In OWL-POS, 67 classes are used to define the POS components. All classes and their inheritances are shown in Fig. 3 except subclasses of $EnumDataType$.

We design an inheritance structure of classes for components. In POSs, partitions, the health monitor (a set of configured tables, e.g., $MultiPartitionHMTable$, $PartitionHMTable$), and interpartition communication components (e.g., $Port$, $Channel$) are statically configured at system build-time and initialized during system/partition initialization. Processes and intrapartition communication components (e.g., $Buffer$, $Semaphore$, $Blackboard$, $Event$) are dynamically created at system runtime. Two classes $ConfiguredThing$ and $CreatedThing$, which are subclasses of $Thing$, are used to model these two categories of components, respectively. A fragment of OWL-POS in the Manchester syntax is as follows:

$$\textbf{Class}: Event\ \ \textbf{SubclassOf}: CreatedThing$$

$$\textbf{Class}: CreatedEvent\ \ \textbf{SubclassOf}: Event$$

$$\textbf{Class}: Port\ \ \textbf{SubclassOf}: ConfiguredThing.$$

OWL2 does not provide a primitive type to represent enumerations. In this paper, a class and a set of individuals are defined in OWL-POS for each enumeration data type used in ARINC 653. Possible values of an enumeration are instances of this class and are represented as individuals. All of these classes are subclasses of $EnumDataType$. For instance, the data structure operating mode is an enumeration. Four individuals ($COLD\_START$, $WARM\_START$, $NORMAL$, and $IDLE$) and a class $OperatingModeType$ are defined to express the operating mode of partitions as follows:

$$\textbf{Class}: OperatingModeType\ \ \textbf{SubclassOf}: EnumDataType$$

$$\textbf{EquivalentTo}: \{COLD\_START, IDLE, NORMAL, WARM\_START\}.$$

B. Relations

The relations between components and those between components and their attributes are modeled by about 150 object/data type properties in the OWL-POS. Each property has a domain and a range.

We design an inheritance structure of properties in OWL-POS according to the inheritance structure of classes. The category of properties, $configuedObjectProperty$, models the relations between two configured classes. The category of properties, $createdObjectProperty$, models the relations of created classes. The $createdObjectProperty$ is further categorized into two subproperties: $fixedObjectProperty$ and $variableObjectProperty$. The subproperties of the first category cannot be changed after being created. The subproperties of the second category can be changed at system run-time.

For the properties, OWL2 has the functional restriction to define a property, which can be seen as a special case of cardinality constraints. A property R is functional, if every individual in the domain of R is connected to at most one individual in the range of R. It does not require every individual to have a corresponding individual. Thus, a functional property is actually a partial function. In POSs, relations between

TABLE I
TYPICAL COMPOSED CONSTRAINTS IN OWL-POS

No.	Constraints (Axioms) in OWL-POS	Description	Invariants in Event-B
(1)	$partition\_has\_OperatingMode$ **some** $(\{NORMAL\})$ **SubClassOf** : $partition\_has_Processes$ **min** 1	Partitions in $NORMAL$ operating mode have at least one created process.	$\forall part \cdot (part\_mode(part) = PM\_NORMAL \Rightarrow proc\_of\_part^{-1}[\{part\}] \neq \varnothing)$
(2)	$partition\_has\_OperatingMode$ **some** $(\{IDLE\})$ **SubClassOf** : $partition\_has\_Processes$ **max** 0	Partitions in $IDLE$ operating mode have no created processes.	$\forall part \cdot (part\_mode(part) = PM\_IDLE \Rightarrow part \notin ran(proc\_of\_part))$
(3)	$partition\_has_Processes$**some** $(process\_has_State$ **some** $(\{READY, RUNNING\}))$ **SubClassOf** : $partition\_has\_OperatingMode$ **some** $(\{NORMAL\})$	Partitions in which there are created processes in $READY$ and $RUNNING$ states are in $NORMAL$ operating mode.	$\forall proc \cdot ((proc\_state(proc) = PS_{Ready} \vee proc\_state(proc) = PS\_Running) \Rightarrow part\_mode(proc\_of\_part(proc)) = PM\_NORMAL)$
(4)	$partition\_has\_LockLevel$ **some** $unsignedInt[<1]$ **SubClassOf** : $partition\_has\_OperatingMode$ **some** $(\{NORMAL\})$	Partitions whose lock levels are less than 1 are in $NORMAL$ operating mode.	$\forall p.(locklevel\_of_{part(p)<1} \Rightarrow part\_mode(p) = PM\_NORMAL)$
(5)	$is\_a\_Proc\_wf\_QueuingPort$ **min** 1 **SubClassOf** : $process\_has\_State$ **some** $(\{WAITING\})$	Processes which are waiting for a queuing ports are in $WAITING$ state.	$\forall port, p.(p \in dom(procs\_wf\_qports(port)) \Rightarrow proc\_state(p) = PS_{Waiting})$

components are usually total functions. This means that each individual in the domain class of a property has exactly one corresponding individual in the range class. For instance, object property $is\_a\_Process\_of\_Partition$ defines the relation from the class $CreatedProcess$ to $Partition$. Since each created process should belong to a partition, this property should be a total function. For functional property R, where C and C' are the domain and range of R, we add an equivalence "**EquivalentTo** : R **min** 1 C'" to the class C to represent that R is a total function.

It is notable that due to the spatial and temporal partitioning of POSs, if there is a relation between two components, the relation or its inversion is a functional relation. Therefore, each property or its inversion in OWL-POS is a functional property. For convenience, we define two object properties between two components. One of them is the inversion of the other. A fragment of OWL-POS in the Manchester syntax is as follows:

ObjectProperty : $is\_a\_Process\_of\_Partition$
Domain : $CreatedProcess$ **Range** : $Partition$
Characteristics : $Functional$
SubPropertyOf : $fixedObjectProperty$
InverseOf : $partition\_has\_Processes$
Class : $CreatedProcess$
EquivalentTo : $is\_a\_Process\_of\_Partition$
min 1 $Partition$.

C. Constraints

The relations in the previous subsection define the restrictions between two classes/data types using properties. There are restrictions between classes that cannot be specified by one property. These restrictions are called constraints in this paper. The constraints are categorized into simple constraints, which are constraints on classes, and composed constraints, which are constraints on object/data type properties.

Simple constraints define the relation between classes that cannot be represented by object properties. We use the expressions $Union$, $Intersection$, $Disjoint$, $Subclass$, and $Equivalence$ in simple constraints. A fragment of OWL-POS of simple constraints is as follows:

Class : $SourcePort$ **SubClassOf** : $Port$
DisjointWith : $DestinationPort$
EquivalentTo : $QueuingSrcPort$
or $SamplingSrcPort$.

Composed constraints are expressed as general axioms in OWL-POS. Sixteen composed constraints are defined. Typical composed constraints and their descriptions are shown in columns 2 and 3 of Table I.

V. TRANSLATION INTO EVENT-B

After presenting OWL-POS, this section introduces the approaches to translating the ontology and APEX services into Event-B.

A. Translating OWL-POS Into Event-B

The basic mapping in the OWL2EB is shown in Table II. Due to the inductive verification in Event-B, we can use abstract sets and relations in Event-B to represent the structural part of ARINC 653 and use the axioms and invariants to constrain them. The state-space problem of formal verification is resolved by induction. In OWL-POS, each property or its inversion is a functional property, which are both modeled. We use partial and total functions in Event-B to represent the OWL-POS properties. For simplicity of the Event-B specification, we only translate functional properties into Event-B. This decision makes the Event-B specification more concise. Functional properties which are not total functions are translated into partial functions ($\nrightarrow$ in

TABLE II
MAPPING OWL-POS (DL) INTO EVENT-B (FOL)

OWL-POS Components	Mapping in Event-B
Thing, **Nothing**	$UNIV$, ϕ
primitive data type (int, bool, etc.)	primitive data type in Event-B
Enum data type	**CONSTANTS** $o_1 \ldots o_n$, **SET** C, $Partition(C, \{o_1\}, \ldots, \{o_n\})$
Composed data type	**SET** C
ConfiguredThing C	**SET** C
CreatedThing C	**VARIABLE** C
configured Object/Data Property R	**CONSTANT** R,
Domain : C **Range** : C'	
Characteristics : $Functional$	**AXIOM** $R \in C \nrightarrow C'$
***EquivalentTo** : R **min** 1 C'	***AXIOM** $R \in C \rightarrow C'$
created Object/Data Property R	**VARIABLE** R
Domain : C **Range** : C'	
Characteristics : $Functional$	**INVARIANT** $R \in C \nrightarrow C'$
***EquivalentTo** : R **min** 1 C'	***INVARIANT** $R \in C \rightarrow C'$
Simple Constraints: SC	**INVARIANT/AXIOM** SC
C_1 **and** C_2	$C_1 \cap C_2$
C_1 **or** C_2	$C_1 \cup C_2$
C_1 **DisjointWith** : C_2	$C_1 \cap C_2 = \phi$
C_1 **SubclassOf** : C_2	$C_1 \subseteq C_2$
C_1 **EquivalentTo** : C_2	$C_1 = C_2$

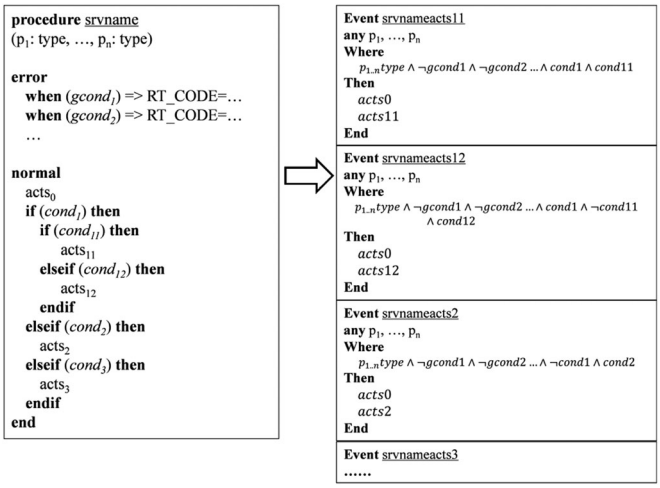

Fig. 4. Semantic mapping from APEX grammar to Event-B.

Event-B). Functional properties with "**EquivalentTo** : R **min** 1 C'" in the declaration of the domain class are translated into total functions ($\rightarrow$ in Event-B).

Constraints in OWL-POS are translated into Invariants or Axioms in Event-B. Since composed constraints are defined on multiple properties, the generated Invariants are manually simplified to improve the verification efficiency in Event-B. Semantics of composed constraints are preserved during this simplification. In column 4 of Table I, the Invariants in Event-B of composed constraints are illustrated.

B. Translating Services Into Event-B

A general structure of a service in the APEX grammar is illustrated in the left part of Fig. 4. The error part describes error handling due to incorrect values of actual input parameters. Although ARINC 653 defines a structured language in APEX grammar to describe the service behavior, we find that ARINC 653 Part 1 only uses compound statements "IF" and "SEQUENCE" to describe complex structures. Moreover, simple actions are described in natural language. Therefore, we design the APEX2EB translation which concentrates on translating the structure of APEX services into events. The detailed behavior of each service represented by simple actions needs to be modeled by hand.

The semantic mapping from APEX grammar to Event-B is illustrated in Fig. 4. Event-B does not have the "IF" compound statement. Therefore, an APEX service should be decomposed into a set of events with non-intersect 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., $acts_{11}$, $acts_{12}$, $acts_2$, $acts_3$) are behaviors under different conditions that do not intersect. Therefore, we use an event to represent the behavior of each body. The type of parameters of the APEX service is encoded as guards of each event ($p_{1..n}type$). In the error part, conditions $gcond_1$, $gcond_2$, etc., indicate incorrect values of parameters, and thus, their negations are translated into Event-B as guards of each event. A simple action, such as "set the specified process state to DORMANT," in Fig. 1, is translated into actions in the event, which is usually represented by a deterministic assignment, e.g., $proc\_state(p) := PS\_Dormant$.

We have designed the APEX2EB algorithm to guide 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

$$c ::= \textbf{ACT}\ act \mid c; c \mid \textbf{IF}\ cond\ \textbf{THEN}\ c \mid \textbf{IF}\ cond\ \textbf{THEN}\ c\ \textbf{ELSE}\ c$$

Algorithm 1: The APEX2EB Translation Algorithm.

```
1  function translate(evts,stmt){
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   }
23   function translate_service(spec){ //spec = ⟨ζ, P, E, S⟩
24     evts ← {⟨ζ, ∅, ∅⟩}
25     evts ← translate(evts, S)
26     assign P to parameters (p) of each event in evts
27     add "∧_i(¬E_i)" to guard list (σ) of each event in evts
28     return {ev | ev ∈ evts ∧ ev.α ≠ ∅}
29   }
```

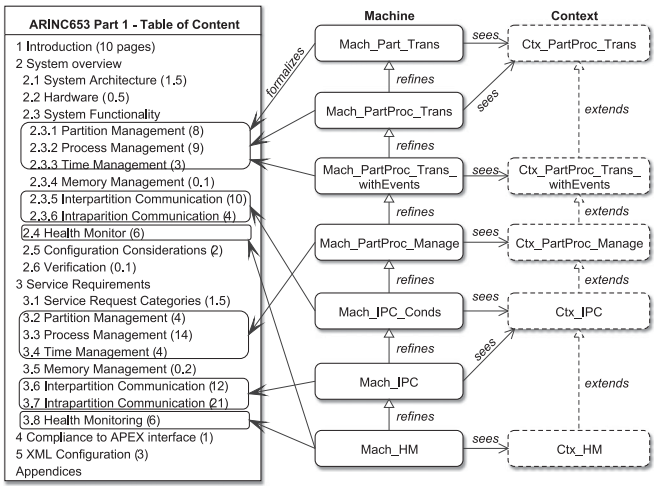

Fig. 5. ARINC 653 Part 1 and the Event-B specification.

where "$\mathbf{ACT}\ act$" is a simple action, and "$c; c$" is the sequential statement.

The algorithm translates a service requirement presented in this syntax into a set of events. The function $translate$ translates a statement $stmt$ into a set of events. An event in the $evts$ set is a tuple $\langle \iota, p, \sigma, \alpha \rangle$, where 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. The translation of simple actions and compound statements ";" is straightforward. For each "IF" statement, we duplicate $evts$ into $evts'$, and add the "IF" conditions as guards to each event in $evts$. Then, we add the negated condition to each event in $evts'$, therefore obtaining all possible nonintersecting guards for both "IF." The function $translate\_service$ translates a service requirement $spec$ into the final set of events by invoking $translate$. A service requirement is a tuple $\langle \zeta, P, E, S \rangle$, where ζ is the service name, P is an input parameter list, E are error conditions in the error part, and S is a statement of the service's normal part. Then, $translate\_service$ assigns the parameters of the service to the parameter list of each event, and adds the negation of the conditions in the error part to the guard list of each event. Finally, it removes those events with empty actions. The event name in the final event set is manually renamed according to its meaning.

VI. Event-B Specification of POSs

This section first discusses the refinement structure of the Event-B specification. Then, some fragments of Event-B and the invariants are illustrated. Finally, the formal verification approach on the Event-B specification is presented.

A. Structure of Formal Specification

The document structure of the ARINC 653 standard and the verification objectives are the major factors to be considered when designing the specification structure.

The complete document structure and the number of pages of each section of ARINC 653 Part 1 are shown in the left part of Fig. 5. The content of ARINC 653 standard is divided into five parts: overview, system functionality, service requirements, configuration, and verification. The system functionality (including health monitoring) and service requirements are the main parts of the standard. Section 1 and subsections 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. The configurations described in subsection 2.5 and section 5 define the information and data format of the system configuration for integration and deployment. Note that the subsection 2.3.4 memory management in ARINC 653 does not define any service.

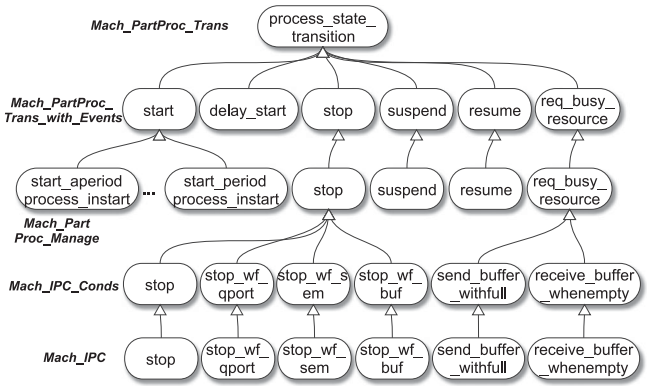

Fig. 6. Excerpt of events and their refinement.

According to the document structure and the verification objectives discussed in Section III, we design the structure of the formal specification as shown in the right part of Fig. 5. We first formalize the system functionality of partition, process, and time management. The Event-B machine $Mach\_Part\_Trans$ models the partition operating modes. $Mach\_Part\_Proc\_Trans$ refines the partition operating modes and adds process state transitions. $Mach\_Part\_Proc\_Trans\_withEvents$ and $Mach\_PartProc\_Manage$ formalize all the actions in the system functionality and the service requirements, respectively. Then, the system functionality and service requirements of the communication are added: $Mach\_IPC\_Conds$ specifies the functionality and actions of interpartition and intrapartition communication, and $Mach\_IPC$ formalizes their service requirement. Finally, the system functionality and service requirements of the health monitor are formalized in $Mach\_HM$. The Event-B contexts are also designed according to the refinement structure. The Event-B models generated in the previous sections are moved into corresponding contexts or machines.

B. Events and Refinements

All services in ARINC 653 have been translated into Event-B as events. In order to verify consistency between the system functionality and the services, a set of events are designed according to the actions on POS components defined in the system functionality of ARINC 653. They represent the pre- and postcondition of each action, which are used to check whether the services correctly refine the actions. The structure of event refinement is partially illustrated in Fig. 6, where the event $process\_state\_transition$ is stepwise refined.

In order to clearly present the events in the specification, examples of events are illustrated in this subsection. The

semantics of an event are introduced in [5, Ch. 5] in detail. Unlike general-purpose OSs, process state transitions in POSs are complex, since process states are dependent on partition operating modes. Process states and transitions are encoded in Event-B in the machine $Mach\_PartProc\_Trans$ as follows. From the guard of the $process\_state\_transition$ event (**grd27**), where "$\Rightarrow$" is the logical implication, we can see the nesting between partition operating modes and process states.

Process state transitions in Event-B only model the possible transition path, not the actions of the process control to trigger the transitions. The actions are modeled in machine $Mach\_PartProc\_Trans\_withEvents$ as the events $suspend$, $resume$, $stop$, $start$, $req\_busy\_resouce$, etc. In these events, we strengthen the guard of $process\_state\_transition$. The event $req\_busy\_resouce$ is defined as follows. **grd07** and **grd27** in $process\_state\_transition$ are strengthened by **grd04**, **grd05**, and **grd06**. The event $req\_busy\_resouce$ is refined in $Mach\_PartProc\_Manage$ and then extended in $Mach\_IPC\_Conds$.

process_state_transition $\widehat{=}$ **any** $part\ proc\ newstate$
where
@grd01 $part \in PARTITIONS$
@grd02 $proc \in processes$
@grd03 $newstate \in PROCESS\_STATES$
@grd06 $proc\_of\_part(proc) = part$
@grd07 $part\_mode(part) \neq PM\_IDLE$
..... //some guards are omitted here
@grd27 $(part\_mode(part) = PM\_NORMAL \wedge proc\_state(proc) = PS\_Running) \Rightarrow (newstate = PS\_Running \vee newstate = PS\_Ready \vee newstate = PS\_Waiting \vee newstate = PS\_Suspend \vee newstate = PS\_Dormant)$
then
@act01 $proc\_state(proc) := newstate$
end
req_busy_resource refines process_state_transition $\widehat{=}$
any $part\ proc\ newstate$
where
@grd01 $proc \in processes$
@grd02 $newstate \in PROCESS\_STATES$
@grd03 $proc\_of\_part(proc) = part$
@grd04 $part\_mode(part) = PM\_NORMAL$
@grd05 $proc\_state(proc) = PS\_Running$
@grd06 $newstate = PS\_Waiting$
then
@act01 $proc\_state(proc) := newstate$
end

C. Invariants

Invariants in the Event-B specification represent the safety properties of POSs. They have two parts: invariants generated from OWL-POS and invariants manually modeled according to ARINC 653. Event-B uses FOL and set theory, while the foundation of OWL2 is DL which is a fragment of FOL. Therefore, some constraints of ARINC 653 can be represented in Event-B, but not in OWL-POS. Besides the constraints in OWL-POS, we have extracted a set of constraints from ARINC 653 and modeled them in the Event-B specification as invariants. For instance, a semaphore in ARINC 653 has a maximum value. The current value of the semaphore must not be larger than the maximum value. This constraint is not modeled in OWL-POS. We represent it as an invariant in Event-B $\forall p \cdot (value\_of\_semaphores(p) \leq MaxValue\_of\_Semaphores(p))$. In the Event-B specification, 95 invariants are specified and verified for ARINC 653.

D. Formal Verification Approach

The main verification approach in Event-B is formal reasoning of proof obligations, which must be proved to show that machines have their specified properties. Formal definitions of all proof obligations are given in [5]. In order to verify the invariants preservation and the refinement, proof obligations of invariants preservation, guard strengthening, and simulation are mainly used in this paper.

Invariant preservation states that invariants are maintained whenever variables change their values. 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, it is also the corresponding abstract one. 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 action is not contradictory with the action of the corresponding abstract event.

For instance, for an event E as shown in (1) and an invariant $inv(s, c, v)$, the generated proof obligation of the invariant on E is as follows:

$$A(s, c), I(s, c, v), G(s, c, v, x), BA(s, c, v, x, v') \vdash inv(s, c, v')$$

where $A(s, c)$ is the axioms seen in the machine, $I(s, c, v)$ is the invariants of the machine, and $inv(s, c, v) \in I(s, c, v)$. All these proof obligations can be generated by the proof-obligation generator in RODIN and proved by automatic provers, SMT solvers, and interactive provers.

VII. Results and Discussion

A. Specification and Proof Statistics

Table III shows the statistics of the OWL-POS ontology and the Event-B specification. The third column of (b) shows the numbers of Event-B elements (80% in total) that are translated from OWL-POS. Since ARINC 653 does not define the runtime information of scheduling (e.g., the current executing process), process management (e.g., the deadline time of a process), etc., we have to manually model them in the Event-B specification for the completeness.

Table IV shows the statistics of formal analysis in RODIN. The lines of code (LOC) of the machines increases gradually

TABLE III
STATISTICS OF FORMAL SPECIFICATION

(a) OWL-POS

Metrics	Number
Class	67
Individual	51
Object Property	91
Data Property	57
Constraint	26

(b) Event-B Specification

Metrics	Number	# of Translated
Set	28	28
Constant	83	71
Variable	58	47
Axiom	69	42
Invariant	95	79
Total	333	267 (80%)

TABLE IV
STATISTICS OF FORMAL ANALYSIS

Machine	LOC	POs	Automatically Proved	Interactively Proved
Mach_Part_Trans	30	6	6 (100%)	0 (0%)
Mach_PartProc_Trans	223	128	122 (95%)	6 (5%)
Mach_PartProc_Trans_withEvents	474	214	212 (99%)	2 (1%)
Mach_PartProc_Manage	1174	618	609 (99%)	9 (1%)
Mach_IPC_Conds	2405	382	380 (99%)	2 (1%)
Mach_IPC	2588	309	309 (100%)	0 (0%)
Mach_HM	3056	15	15 (100%)	0 (0%)
Total	3056	1672	1653 (99%)	19 (1%)

since the refinement machine is an extension of the refined machine. More than 1600 proof obligations (POs) are automatically generated in RODIN and 99% of them are automatically discharged.

B. Errors Found in ARINC 653 and Their Hazards

During formal analysis of the Event-B specification, we find six errors in ARINC 653 Part 1. The errors cause an incorrect specification for process management and communications and, hence, malfunctions of applications. The consequences of the errors are not allowed in safety-critical systems. All these errors have been fixed in the Event-B specification.

1) Process State Transitions: An incomplete description of process state transitions in the system functionality of ARINC 653 is detected. The errors are shown as follows.

1) E1: The service requirement of the *RESUME* service indicates that in the COLD/WARM_START mode, a suspended period process can be resumed and its state transits from Waiting to Waiting. This action is missing in the "Waiting → Waiting" transition conditions in the system functionality.
2) E2: The service requirement of the DELAYED_START service indicates that if an aperiodic process is delayed_started (if the delay time > 0) in the NORMAL mode, its state transits from Dormant to Waiting. This action is missing in the "Dormant → Waiting" transition conditions in the system functionality.
3) E3: The service requirement of the *DELAYED_START* service also indicates that if an aperiodic process is *delayed_started* (if the delay time $= 0$) in the NORMAL mode, its state transits from Dormant to Ready. This action is missing in the "Dormant → Ready" transition conditions in the system functionality.

The incompleteness above is found by verifying the guard strengthening between machine $Mach\_PartProc\_Trans\_withEvents$ and its refinement machine $Mach\_PartProc\_Manage$.

2) Process Management Services (E4): The error is in the service requirement of the *RESUME* service. In fact, suspending an aperiodic process that has been *delayed_started* causes transition into Waiting state of the process. When that process is resumed, it should be retained in the Waiting state if the delay time has not been reached. But in the *RESUME* service, the aperiodic process is set into the Ready state. 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) Communication Services: We find two errors in the requirements of the communication services.

1) E5: The error is in the $SEND\_QUEUING\_MESSAGE$ service in interpartition communication. The service is used to send a message via a specified queuing port. The system functionality requires that if there is not enough space in the queuing port to accept the message, the process is blocked and stays in the waiting queue until the specified time-out expires (in this case, the message is lost) or space becomes free (in this case, the message is inserted into the port's message queue). However, in the service specification when space becomes free before the expiration of the time-out, the sent message is not inserted into the message queue. This error is found by verifying the simulation of the events $send\_queuing\_message\_needwait$ and $wakeup\_waitproc\_on\_srcqueports$ between the machines $Mach\_IPC\_Conds$ and $Mach\_IPC$.
2) E6: The error is in the *RECEIVE_BUFFER* service in intrapartition communication. This service is used to receive a message from a specified buffer. The system functionality requires when the buffer is not empty, then the receiving process can receive a message directly, and the message has to be removed from the message queue of this buffer. But in the service specification, we find that the received message is not removed. This error is found by verifying the simulation of the event $receive\_buffer$ between the machines $Mach\_IPC\_Conds$ and $Mach\_IPC$.

C. Validation of Open-Source POSs

The incompleteness of process state transitions are errors about incomplete description of ARINC 653 itself. Thus, we only manually validate the other three errors in XtratuM and POK, which implement the ARINC 653 services, discovering that one of these errors exists in the validated POSs.

XtratuM is a separation hypervisor aiming at providing a framework to run several OSs in a partitioned environment. The version of XtratuM we validate is v3.7.3 for SPARC v8 architecture. Since XtratuM is a hypervisor, the processes/threads in partitions are managed by guest OSs. Therefore, XtratuM does

not implement the *RESUME* and *SEND_BUFFER* services. On the other hand, the syscall *SendQueuingPort* is used to implement the *SEND_QUEUING_MESSAGE* service, but the syscall does not implement the *TIME_OUT* parameter as specified in the service. Due to the incomplete implementation of ARINC 653 in XtratuM, errors **E4**, **E5**, and **E6** are not found in XtratuM.

POK is a POS for safety-critical systems. The version of POK we validate is the latest one released in 2014. POK implements all APEX services. The syscall *pok_thread_resume* is used to implement the *RESUME* service. In this syscall, the state of a resumed thread is set to *POK_STATE_RUNNABLE* (i.e., *PS_Ready* in this paper) without any judgment. Therefore, the error **E4** exists in POK. The errors **E5** and **E6** are not found in POK.

D. Discussion

1) Effectiveness of the OWL-POS Ontology: Due to the RDF/XML-based formation of OWL2, the interoperability of OWL-POS is guaranteed. OWL-POS reduces not only the efforts of developing the Event-B specification by translation, but also the cost of formal verification. Except the events which are translated from service requirements of ARINC 653, 80% of Event-B elements are translated from OWL-POS. Moreover, OWL-POS representation of ARINC 653 regulates the generated sets, axioms, and invariants in Event-B. Thus, the degree of automatic verification of the Event-B specification is increased. In our previous paper [25], we have developed an Event-B specification of ARINC 653 according to the standard directly. The degree of automatic verification of Event-B proof obligations is 83%. By integrating the ontology technology and Event-B, the number of interactively proved proof obligations decreases from 300 to 19. Thus, the degree of automatic verification of Event-B proof obligations is promoted to 99%. It is certain that the manual creation of the ontology using the Protégé editor is quite time-consuming and the usage of the OWL-POS ontology leads to an increment of the design cycle time. However, the creation of OWL-POS is a one-time process and OWL-POS is reusable.

2) Completeness of Event-B Specification: Although we have formalized all of ARINC 653 services and the system functionality, some implementation-related details of POSs are not specified in ARINC 653 Part 1 and not covered in our formalization.

1) Since ARINC 653 specifies nothing about the booting process, the booting/initialization of the POSs is not modeled in the Event-B specification.
2) Since the initialization of partitions is not specified in ARINC 653, only an abstract specification of the partition initialization is defined in the Event-B specification.
3) Partition switch is not specified in ARINC 653 and it is omitted in the Event-B specification.

For the purpose of verifying the ARINC standard or ARINC-based applications, these eliminations do not affect the verification result. However, these details should be implemented when formally developing POSs from the ARINC 653 standard.

3) Reusability of the Formal Specification: The Event-B specification of ARINC 653 defines the abstract specification of POSs. The specification can be used in the model-based development and the verification of POSs. Composition of the Event-B specification and application models enables the simulation, analysis, and verification for IMA systems. On the other hand, the Event-B specification provides the possibility of formally developing a new POS by refinement and further automatic code generation. The RODIN environment provides many code generation plug-ins for C, C++, Java, and Ada. The difficulty of code generation 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. It requires that the Event-B specification is detailed enough and the code generation in RODIN need to be revised accordingly.

VIII. Conclusion and Future Work

In this paper, we have presented a general methodology for formalizing informal standards or requirements by integrating ontology and refinement. The OWL-POS ontology for POSs has been developed as the intermediate model between ARINC 653 and Event-B. By semi-automatic translation from the APEX grammar and OWL-POS into Event-B, a complete Event-B specification for ARINC 653 compliant POSs has been developed in this paper. During formal analysis, a few significant problems in ARINC 653 have been discovered. We have also found some errors in open-source POSs by validation. The formal specification could be a foundation in the life cycle of POSs and IMA applications. The proposed methodology is applicable to formalization and verification of systems in accordance with informal standards and requirements.

In future work, since ARINC 653 is being considered to support multicore platform, we will develop formal specification of multicore POSs. Second, we consider to refine the formal specification in this paper to a low-level design and develop verification approaches to formally analyze ARINC 653 compliant POSs on source-code level.

References

[1] *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1—Required Services*, Aeronautical Radio, Inc., Annapolis, MD, USA, Nov. 2010.
[2] M. Fraser, K. Kumar, and V. Vaishnavi, "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Trans. Softw. Eng.*, vol. 17, no. 5, pp. 454–466, May 1991.
[3] G. Klein *et al.*, "Comprehensive Formal Verification of an OS Microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 1–70, Feb. 2014.
[4] *OWL 2 Web Ontology Language Primer*, 2nd ed., W3C, Cambridge, MA, USA, Dec. 2012.
[5] J. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge Univ. Press, 2013.
[6] W. Dai, V. N. Dubinin, and V. Vyatkin, "Automatically generated layered ontological models for semantic analysis of component-based control systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2124–2136, Nov. 2013.
[7] A. Iliasov *et al.*, "Developing mode-rich satellite software by refinement in Event-B," *Sci. Comput. Program.*, vol. 78, no. 7, pp. 884–905, Jul. 2013.
[8] D. Méry and N. K. Singh, "Formal specification of medical systems by proof-based refinement," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1, pp. 1–25, Jan. 2013.
[9] XtratuM Hypervisor. [Online]. (2015) Available: http://www.xtratum.org/

[10] POK. [Online]. (2015) Available: http://pok.tuxfamily.org/
[11] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.
[12] G. Klein, "Operating system verification—An overview," *Sadhana*, vol. 34, no. 1, pp. 27–69, Feb. 2009.
[13] F. Verbeek *et al.*, "Formal specification of a generic separation kernel," *Arch. Formal Proofs*, 2014. Available at: http//www.isa-afp.org/entries/CISC-Kernel.shtml
[14] D. Sanán, A. Butterfield, and M. Hinchey, "Separation kernel verification: The Xtratum case study," in *Verified Software: Theories, Tools and Experiments (ser. LNCS)*, vol. 8471. Berlin, Germany: Springer-Verlag, 2014, pp. 133–149.
[15] F. Verbeek *et al.*, "Formal API Specification of the PikeOS separation kernel," in *NASA Formal Methods (ser. LNCS)*, vol. 9058. Berlin, Germany: Springer-Verlag, 2015, pp. 375–389.
[16] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev, "Proving memory separation in a microkernel by code level verification," in *Proc.14th IEEE Int. Symp. Object/Compon./Service-Oriented Real-Time Distrib. Comput. Workshops*, Newport Beach, CA, USA, Mar. 2011, pp. 25–32.
[17] S. Tverdyshev, "Extending the GWV security policy and its modular application to a separation kernel," in *NASA Formal Methods (ser. LNCS)*, vol. 6617. Berlin, Germany: Springer-Verlag, 2011, pp. 391–405.
[18] R. Richards, "Modeling and security analysis of a commercial real-time operating system kernel," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. New York, NY, USA: Springer, 2010, pp. 301–322.
[19] M. Wilding, D. Greve, R. Richards, and D. Hardin, "Formal verification of partition management for the AAMP7G microprocessor," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. New York, NY, USA: Springer, 2010, pp. 175–191.
[20] A. Gomes, "Formal specification of the ARINC 653 architecture using circus," M.Sc. thesis, Dept. Comput. Sci., Univ. York, York, U.K., 2012.
[21] Y. Wang, D. Ma, Y. Zhao, L. Zou, and X. Zhao, "An AADL-based modeling method for ARINC653-based avionics software," in *Proc. IEEE 35th Annu. Comput. Softw. Appl. Conf.*, Munich, Germany, Jul. 2011, pp. 224–229.
[22] J. Delange, L. Pautet, and F. Kordon, "Modeling and Validation of ARINC653 architectures," in *Proc. Embedded Real Time Softw. Syst. Conf.*, Toulouse, France, May 2010, pp. 1–8.
[23] F. Singhoff and A. Plantec, "AADL modeling and analysis of hierarchical schedulers," *ACM SIGAda Ada Lett.*, vol. 27, no. 3, pp. 41–50, Dec. 2007.
[24] P. Cámara, J. Castro, M. Gallardo, and P. Merino, "Verification support for ARINC-653-based Avionics software," *Softw. Test. Verif. Rel.*, vol. 21, no. 4, pp. 267–298, Jan. 2011.
[25] Y. Zhao, Z. Yang, D. Sanán, and Y. Liu, "Event-based formalization of safety-critical operating system standards: An experience report on ARINC 653 using Event-B," in *Proc. Int. Symp. Softw. Rel. Eng.*, Nov. 2015, pp. 281–292.

Yongwang Zhao received the B.S. degree in information systems from the Beijing Information Technology Institute, Beijing, China, in 2002, and the Ph.D. degree in computer science from Beihang University, Beijing, in 2009.

In 2009, he joined the School of Computer Science and Engineering, Beihang University, as an Assistant Professor. He has also been a Research Fellow with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, since 2015. His research interests include formal methods, real-time operating systems, and architecture analysis and design language.

Dr. Zhao is a senior member of the China Computer Federation and a member of the Association of Computing Machinery.

David Sanán received the M.S. degree in computer science, and the Ph.D. degree in software engineering and artificial intelligence from the University of Málaga, Málaga, Spain, in 2003 and 2009, respectively.

He was a Research Fellow at the Singapore University of Technology and Design, Singapore; Trinity College Dublin, Dublin, Ireland; and the National University of Singapore, Singapore. In 2015, he joined Nanyang Technological University, Singapore, where he is currently a Research Fellow. His research interests include formal methods and, in particular, the verification of software. In the past, he developed techniques for the verification of software using model checking. His current research topic is in the formalization and verification of separation microkernels aiming multicore architectures.

Fuyuan Zhang received the B.S. degree in information security from the Beijing University of Posts and Telecommunications, Beijing, China, in 2006; the M.Eng. degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2009; and the Ph.D. degree in computer science from the Technical University of Denmark, Kongens Lyngby, Denmark, in 2012.

He is currently a Research Fellow with the School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interests include formal verification of IT systems, computer security, and quantum computation.

Yang Liu received the bachelor's and Ph.D. degrees in computer science from the National University of Singapore (NUS), Singapore, in 2005 and 2010, respectively.

He was a Post-Doctoral Researcher at NUS. In 2012, he joined Nanyang Technological University, Singapore, as an Assistant Professor. His research interests include software engineering, formal methods, and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.