Hindawi
Scientific Programming
Volume 2022, Article ID 2079880, 26 pages
https://doi.org/10.1155/2022/2079880

Research Article

A Comprehensive Formalization of AADL with Behavior Annex

Yu Tan,[1] **Yongwang Zhao**,[2] **Dianfu Ma**,[3] **and Xuejun Zhang**[1]

[1]*Beijing Institute of Control and Electronic Technology, Beijing 100038, China*
[2]*College of Computer Science and Technology, School of Cyber Science and Technology, Zhejiang University, Hangzhou 310058, China*
[3]*State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100191, China*

Correspondence should be addressed to Xuejun Zhang; ty138686@sina.com

Received 20 October 2021; Revised 22 November 2021; Accepted 8 December 2021; Published 4 January 2022

Academic Editor: Punit Gupta

In safety-critical fields, architectural languages such as AADL (Architecture Analysis and Design Language) have been playing an important role, and the analysis of the languages and systems designed by them is a challenging research topic. At present, a formal method has become one of the main practices in software engineering for strict analysis, and it has been applied on the tools of formalization and analysis. The formal method can be used to find and resolve the problems early by describing the system with precise semantics and validating the system model. This article studies the comprehensive formal specification and verification of AADL with Behavior annex by the formal method. The presentation of this specification and semantics is the aim of this article, and the work is illustrated with an ARINC653 model case study in Isabelle/HOL.

1. Introduction

In safety-critical domains such as avionics, aerospace, automotive, and defence, a latent software error even can give rise to catastrophic consequences. Such systems have to be carefully designed and analyzed according to some strict standards such as DO-178C [1], which stipulates analysis, testing, and certification activities. Formal methods have become the recommended practice in the safety-critical fields. Formal methods are special techniques based on mathematics and are suitable for the description, development, and verification of software and hardware systems. By applying formal methods to software and hardware designs, it is hoped that, like other engineering disciplines, appropriate mathematical analysis can be used to improve the reliability and robustness of designs. In the design of computer software systems, formal verification means that mathematical methods can be used to prove their correctness or incorrectness according to one or some formal specifications or attributes.

Theorem proving, program analysis, and model checking are the main branches of formal verification. To be formally verified, systems should be firstly specified with a specific formalism. AADL (Architecture Analysis and Design Language) [2] is a modeling standard used in safety-critical software engineering to describe the structure of systems, such as a package of software components, which is mapped on an execution platform. AADL adopts formal modeling concepts for the description of software and hardware architecture, so that it is often used to design and analyze the software and execution platform of real-time embedded systems. The operation of these systems depends strongly on meeting nonfunctional requirements such as availability, reliability, responsiveness, throughput, safety, and security. As a supplement of runtime environment in terms of distinct components and their interactions, the standard AADL Behavior annex [3] represents a behavioral extension for AADL, which allows a more detailed specification of the software behavior. Using AADL with its Behavior annex, the complete models can be designed in a way that large information about data models, timing, and communication behaviors is available at the modeling phase, and it is especially effective for the model-driven design of complex embedded real-time systems. AADL is

standardized by the SAE, and its second version was published in 2009 and revised in 2017; for its analyzability and extensibility, AADL has become one of the popular languages within architectural modeling in the industry [4]. AADL has been studied in several projects for different modeling, analysis, simulation, compilation, extension, and formal verification. Moreover, the AADL semantic can be extended via user-defined properties and annexes.

However, AADL cannot be directly generated into the executable code, which is reliably used in safety-critical systems. So how to compile AADL to the C code is our final goal. Although there is not any comprehensive compiler or method from AADL to C on the open-source platform so far, there is some existing related work about the verified compiler or transformation of model languages like Lustre [5], CompCert [6]. As we do, proving the correctness of general-purpose compilers is undeniably a related problem, and fortunately, this work [7] encourages us and presents the possibility of development based on the model-driven design in prover tools. We aim at exploring a formal method of compiling AADL to C-like language, so we firstly limit our survey to work that focuses on the particularities of AADL.

The AADL provides a sufficient syntax and semantic to describe an embedded real-time system based on software/hardware components and their relations. Dealing with such rich models accentuates the need for model analysis and verification. Unfortunately, AADL is a textual and graphical language, which means it is a semiformal modeling language. It lacks formal specification and semantics, and this severely limits both unambiguous communication among model developers, and the development of simulators and formal analysis tools, so itself cannot be directly used for formal verification. In this work, we choose Isabelle/Isar/HOL [8], a tool suite (within its functional language) that gathers specification, validation, and verification of AADL and models, and also the code generation towards AADL runtime C-like language for the future work. The Isabelle/Isar language provides a readable grammar and a convenient way to produce the proofs.

In this article, we provide an approach for the formal verification of behavioral AADL models. In detail, this article makes the following contributions:

(i) Different from transforming AADL into other formal model languages, our work takes an approach by formally specifying AADLs, its corresponding models of definitions, lemmas, and proof structures necessary to verify the model, providing a blueprint for performing similar work in any prover tool. Also we can state and prove a correctness relation between the source and target semantic models, and even directly build on the compilation project in Isabelle/HOL as our future work.

(ii) We consider the AADL focusing on safety-critical software, so our work mostly covers the whole AADL elements including components, communication among components, and Behavior annex defined inside, and also supports the key features and properties. The considered AADL subset consists of both software and hardware AADL components with complex state transitions being comprehensive and that can be used in more realistic applications.

(iii) We perform formal validation and verification of the AADL model and specify the critical properties. Specifically, we (1) analyze and summarize the description for the AADL standard defined by the SAE republished in 2016 and take 47 significant details into account as the grammar rules in Isabelle/HOL; (2) exploit the comprehensive semantics including Behavior annex, Thread, Process, and System, and then integrate them into a whole model execution semantics; and (3) perform formal instantiation, validation, and verification of three realistic AADL models.

In this context, we aim at the comprehensive formal specification and verification of AADL core language (software part) with its Behavior annex. The remainder of this article is organized as follows: in Section 2, we describe the concept of AADL along with its Behavior annex and Isabelle/HOL, and also present the strength of Isabelle/HOL and its specification language to justify why we choose it to model AADL and Isabelle/HOL to perform formal analysis; Section 3 overviews our approach including the AADL elements we selected; in Section 4, we present the syntax of our selection and the validation rules for grammar in Isabelle/HOL; In Section 5, we present the semantics of selection and the verification, and in addition, we present the semantics of Behavior annex; Section 6 then presents a case study; and Section 7 gives the conclusions and future directions.

2. Background

2.1. AADL and Its Behavior Annex. AADL is a textual and graphical language used to model, specify, and analyze architectures (included software and hardware part) of safety-critical and real-time embedded systems, and it has been studied in several projects for different purposes analysis, code generation, extensions, and formal verification. AADL is based on a component-centric model, and it defines the system architecture as a set of interconnected components that hierarchically describes the interfaces, the implementations, the properties, and the channels among components. It describes a system as a hierarchy of software and hardware components and offers a set of predefined component categories as follows:

(i) Software components: data, subprogram, subprogram group, Thread, Thread group, Process, and their types, implementations, features, connections, properties

(ii) Execution platform components (hardware components): processor, memory, bus, and device

(iii) System composites: they represent composite sets of software and execution platform components

(iv) Annex subclauses: they allow annotations expressed in a sublanguage to be attached to the component

and contain Behavior Annex, Error Annex, Data Annex, etc

According to the component categories, AADL software component elements are composed and synchronized to form the whole software system. Figure 1 gives an overview of the AADL software components containing essential constructions.

2.2. Isabelle/HOL Notations. Isabelle/HOL (the full name is Isabelle/Isar/HOL, Isabelle is often for short) is a generic interactive theorem prover for implementing logical formalisms of a specification and verification, and it is the specialization of Isabelle for HOL (higher-order logic) [9]. Isabelle is implemented in ML [10]. This has influenced some of Isabelle/HOL's concrete syntax Isabelle/Isar [11], an extension of Isabelle, which hides the implementation language almost completely. Based on a small (meta)-logical inference kernel, Isabelle's LCF-style architecture ensures very high confidence about its soundness as a theorem prover. Since our whole work focuses on the verification of formalization and the output, moreover, also will invoke the theorem prover's code generator and run the test suite on the C-like code generated by itself in the future, Isabelle is used to prove the methodology in this work. This work mainly restricts itself to the core of Isabelle (simply typed Lambda calculus with ML-style polymorphism and inductive datatypes). The main notations used in this work are explained as follows:

theories, working with Isabelle means creating theories. Roughly speaking, a theory is a named collection of types, functions, and theorems, much like a module in a programming language or a specification in a specification language. In fact, theories in HOL can be either. The general format of a theory T is

theory T

imports $B_1 \ldots B_n$

begin

Declarations, definitions, and proofs

end

where $B_1 \ldots B_n$ are the names of existing theories that T is based on and declarations, definitions, and proofs represent the newly introduced concepts (types, functions, etc.) and proofs about them.

lemma, this command starts the proof and gives the lemma a name. As a result of that final done, Isabelle/HOL associates the lemma just proved with its name. Lemma, theorem, and rule are used interchangeably for propositions that have been proved.

base types, in particular bool, the type of truth values, and nat, the type of natural numbers.

type variables, denoted by 'a, 'b, etc., like in ML.

datatypes, the general form of a datatype definition looks like this:

da tatype $('a_1, \ldots, 'a_n)t = C_1 \text{ " } \tau_1, 1\text{" } \ldots \text{"} \tau_1, n_1\text{"}$

$| \ldots$

$|C_k \text{ " } \tau_k, 1 \text{ " } \ldots \text{" } \tau_k, n_k\text{"}$

It introduces the constructors $C_1 : : \tau_1, 1 \Longrightarrow \ldots \Longrightarrow \tau_1, n_1 \Longrightarrow ('a_1, \ldots, 'a_n)t$ and expresses that any value of this type is built from these constructors in a unique manner.

records, introduces a new record-type scheme by specifying its fields, which are packaged internally to hold up the perception of the record as a distinguished entity. A record of Isabelle/HOL covers a collection of fields, with select and update operations. Here is a simple example:

record point = Xcoord:: int

Ycoord:: int

In this work, we choose a deep embedding method, which does not try to directly represent elements of the language as expressions of the target language (in this case: Isabelle/HOL), but rather encodes them.

3. Approach Overview

The general methodology of our work is given as follows: at first, we select a comprehensive core AADL with its Behavior annex and a policy for modeling as 3.1.1 and 3.1.2 presented; secondly, we define a specification from AADL in Isabelle/HOL and model an example; next, we determine some significant rules about AADL grammar from AADL official standard manual and transform them into the model-checking functions, and then, we present a validation by using these functions; finally, we present the semantics of the comprehensive core AADL and verify some properties (reachability, trace refinement, etc.) by the semantics in Isabelle/HOL. The main idea of our methodology is illustrated in Figure 2.

3.1. Selection of AADL

3.1.1. Selection of Core AADL in This Work. In this article, our work focuses on the specification and analysis of the software components of systems, and most of execution platform components in AADL (virtual processors, memory, buses, virtual buses and devices, etc.) are not under consideration except the processor (simply discussed). Moreover, the group, prototype, and refinement are regarded as a set element, and mainly for the reusability of the AADL code, software systems can be modeled even without these elements; therefore, they are not accounted in this work. This selection of AADL core elements is comprehensive and sufficient to specify and model an embedded system on the software side.

A component can be a subcomponent of the other component. Thus, the following components are supported: *processor*, *data*, *subprogram*, *thread*, *process*, *system*, and thread's *Behavior annex*. A processor component is an abstraction of hardware and software that is responsible for scheduling and executing threads that are bound to it. A data component represents a data type and also static data in the

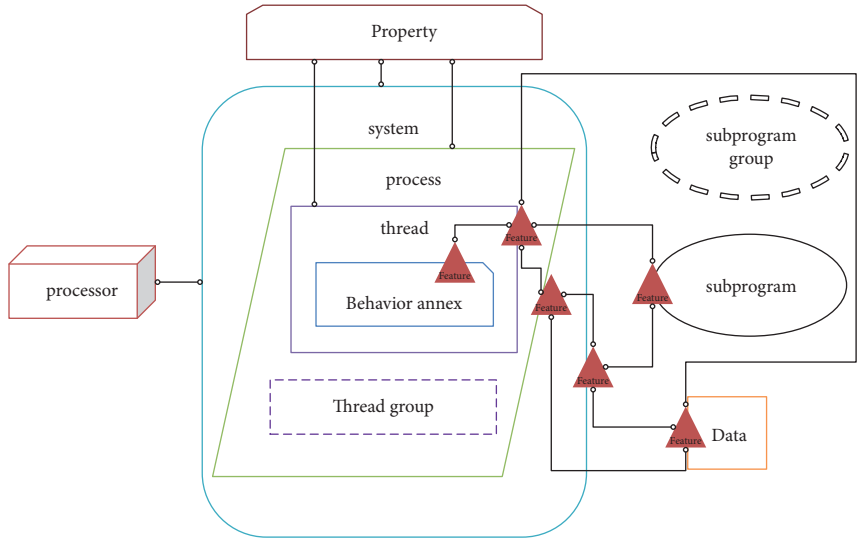

Figure 1: AADL software component elements.

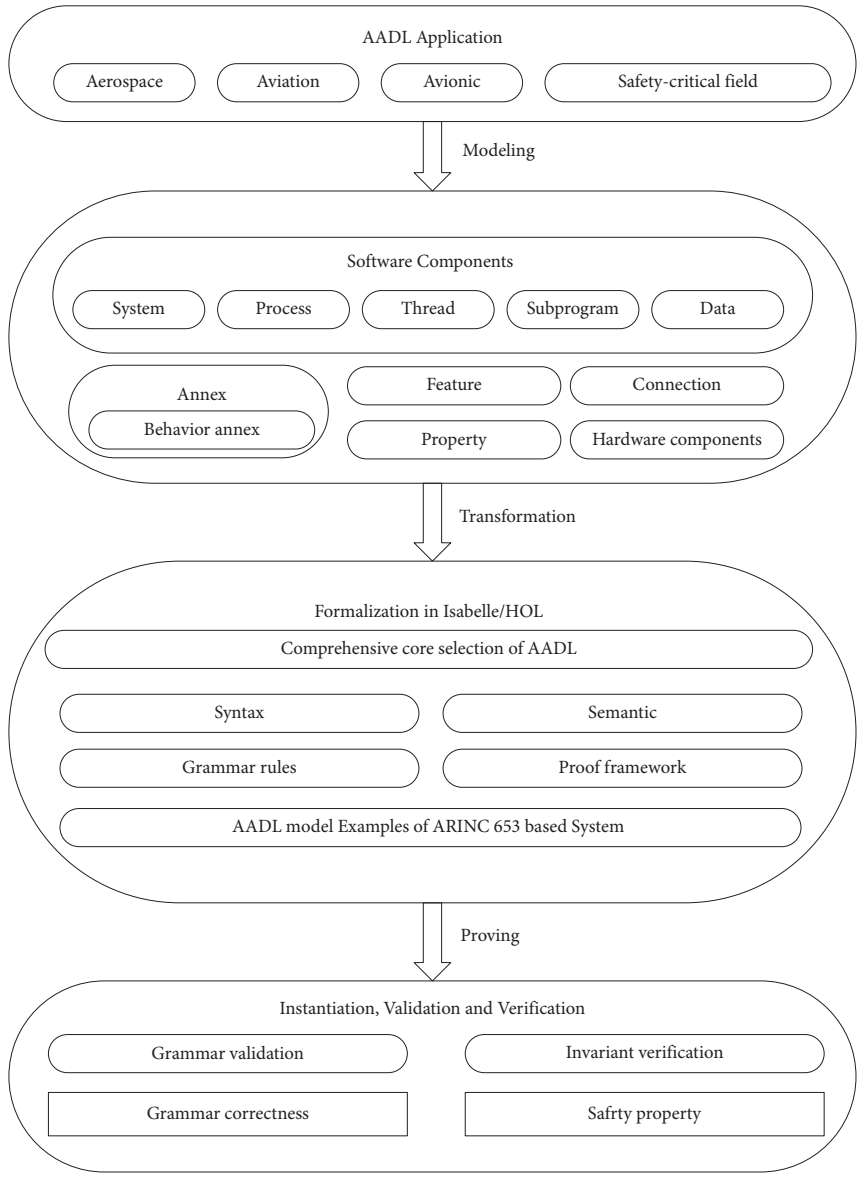

Figure 2: Approach overview.

source text. A subprogram component represents sequentially executed source text that is called with parameters. A thread component is a schedulable unit that is declared within a process component and can be executed concurrently with other threads. A process component represents its virtual address space, and a software system represents an assembly of interacting application software.

A feature is a part of a component definition that specifies how that component interfaces with other components in the system. Features consisted of *port*, *access,* and *parameter* in our work. In addition, features can be combined with properties, and our work can support some temporal and queuing properties, such as Dispatch_Protocol (periodic, sporadic, timed), Period, Queue_Processing_Protocol (FIFO, LIFO), Queue_Size, Elapsed_Time, Execution_Time, and Scheduling_Protocol.

A connection is a linkage between features of two components representing communication of data and control between components. Our work supports connections between *port connections*, *parameter connections*, *dataaccess connections*, and *subprogramaccess connections*.

A property provides information about model elements, and it has a name, a type, and a value. Each property has a value or list of values that is associated with the named property in a given specification. Our work focuses on the indispensable properties, which depend on the specific components.

3.1.2. Selection of AADL Behavior annex. The Behavior annex document provides a standard sublanguage extension to allow behavior specifications to be attached to AADL components. It is an important part of AADL, as it split a whole system model into several single composable components to make design and analysis easier. The Behavior annex of a Behavior annex instance is defined on the vocabulary consisting of its private variables behavior_variable, its states behavior_state, and ports of its parent component. Its transition system is the union of the transitions specified by a behavior_transition. A Behavior Annex specification of a thread contains *variables*, *states*, and *transitions*. The states may be *initial*, *complete*, *execution*, or *final*.

Our work can support the Behavior annex with its specification to enrich the running model. The aim of the Behavior annex is to refine the implicit behavior specifications that are specified by the core of the language. Yet we practically state that behavior specification subclauses can only be added in a thread, and the behavior specification subclauses describe the thread that the behavior specification subclause belongs to, since the execution of the whole system at one processor is actually the execution of one thread.

3.2. Related Work. AADL lacks formal specification and semantics; therefore, it cannot be directly used for formal verification and it is often transformed into several formal model languages to be adopted with existing formal analysis tools. We study several formal approaches on AADL. According to the transformation method and the consideration of the AADL model with or without Behavior annex, these are grouped into three categories.

The first category is often based on model transformation into different languages without (or barely with) Behavior annex such as Petri nets [12], Timed automata [13], TLA+ [14], Lustre [15], Fiacre [16], and CSP [17,18]. These approaches are contented with the AADL semantic described in its standard, which is enough to formally simulate the system and verify a set of behavioral properties. The second category represents work about the model transformation of AADL with its Behavior annex such as BIP [19], Signal [20], TASM [21, 22], and Ocarina [23]; for example, Ref. [19] defines a transformation into the BIP language, and then, the BIP model is transformed into nontimed models to enable model checking and simulation with the BIP framework. The third category almost only specializes in behavior and analysis by using and mapping AADL behavioral models such as IF [24] and real-time Maude [25]. Such a mapping allows the analysis of the performance and the dependability.

These AADL formal approaches mainly consider different AADL subsets (with or without annexes) and carry on formal verification with existing tools such as UPPAAL, Tina, and Polychrony. They often define a model transformation to implement whole AADL model certification instead of AADL itself. Moreover, a formal proof of the semantics preservation of the transformation has not been considered by them. Our work considers several resource information in the transformation, and the theorem prover is used to prove the methodology, that is, the correctness of the translation. The comparisons of the above-related works are listed in Table 1.

These works focus on a subset of the AADL, and most of the related works only consider a small subset of Behavior annex. For AADL elements, our work supports Behavior annex and components, in which variables, states, state transitions of Behavior annex and connections, features, and software components of components are represented by a "+". For the aspect of verified properties, our work considers more types of properties by Isabelle, for example, grammar, reachability, and trace refinement.

The main goal of our work is to contribute to a better integration of the formal techniques in a compilation process. So this article is full of formal approaches revolving around the AADL, and we choose Isabelle/Isar/HOL [8], a tool suite (within its functional language) that gathers specification, validation, and verification of AADL and models, and also the code generation towards AADL runtime C-like language for the future work. Besides the tool suite that can be used as stand-alone compiler, the Isabelle/Isar language provides a readable grammar and a convenient way to produce the proofs.

4. Abstract Syntax and Validation

4.1. Presentation of Abstract Syntax in Isabelle/HOL. This section describes those aspects of components that are common to all AADL component categories. Our work provides the abstract syntax of the considered AADL in

Table 1: Comparison of related AADL formal approaches.

Works	Specification language	Verification tool	Elements covered						Support for transformation correctness verification
			Behavior annex	Components				Safety	
				Thread	Process	System	Others		
Gina et al.	Petri Nets	ADAPT	−	++	−	−	−	−	−
Johnsen et al.	Timed Automata	UPPAAL	+	+++	+	−	+++	+	−
Jean-Francois et al.	TLA+	TLC	−	+	−	−	−	−	−
Jahier et al.	Lustre	Lurette, Lesar	−	++	+	−	−	+	−
Berthomieu et al.	Fiacre	Tina	+	+++	−	−	−	+	−
Yang et al.	CSP	FDR	−	++	−	−	+	++	−
Chkouri et al.	BIP	BIP framework	++	++	−	−	−	+	−
Besnard et al.	Signal	SynDEx	+	+	−	−	−	−	−
Yang et al.	TASM	TASM, UPPAAL	+	+++	+	−	+++	+	−
Mkaouar et al.	LNT	Ocarina	−	+	+	−	+++	+	−
Abdoul et al.	IF	IFx framework	Its own behavioral language	+	−	−	−	++	−
Ölveczky et al.	Real-time Maude	Maude	++	++	−	−	−	+	−
Ours	Isabelle/Isar	Isabelle/HOL	+++	+++	+	+	+++	++	+

Isabelle/HOL: an AADL model contains several software subcomponents (like threads), several features (like ports), and Behavior annex specification. Each Behavior annex can belong to a thread, and each thread with its Behavior annex belongs to a process. In this article, to keep the article reasonably concise, some structural elements and model attributes are expressed in a uniform abstract syntax. In addition, at the whole system level, a system is viewed as a set of processes, and a process is viewed as a set of threads in communication through port and access connections. According to the selection of AADL and its Behavior Annex, the classification of main syntax is discussed in the following subsections.

4.1.1. Features and Connections. Features are a part of component type definition that specifies how that component interfaces with other components in the system. Features are specified as *port*, *access*, and *parameter*. Two components are connected between features by a linkage called connections, and connections can be the transmission of control and data in components' implementation. AADL supports connections between *port*, *access*, and *parameter connections*.

The details of Features and Connections are much more than we have space to present here, and some of them are defined in Figures 3 and 4, and the code snippets are shown in Figures 5 and 6.

4.1.2. Type and Implementation Base. Components represent some hardware or software entity that is part of a system being modeled in AADL. A component has a component type, and zero or more component implementation, which defines a functional interface and realization. The component type acts as the specification of a component that other components can operate against, and the component implementation specifies the realization of a component variant. A component type and implementation instance are presented in Figures 7 and 8.

By default, we consider one component *Type* has only one component *Implementation* to avoid the complexity of the actual modeling arisen from polymorphism. To reduce code redundancy, some basic elements of *Type* and *Implementation*, like *Features* and *Subcomponents*, are declared as the datatypes as the type_base, impl_base, and their own *Properties* to add in the respective components later. Their syntaxes in Isabelle/HOL are presented in Figures 9 and 10.

```
feature ::= port | data_access
          | subprogram_access | parameter
```

Figure 3: The syntax of Feature.

```
connection ::= connection_identifier
               ( port_connection | access_connection | parameter_connection)
               [ '{' { property_association } + '}' ]
               [ in_modes_and_transitions ]
```

Figure 4: The syntax of Connection.

```
datatype ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) Feature =
FPort 'Port | FDataaccess 'Dataaccess | FSubpaccess 'Subpaccess |
FParam 'Parameter
```

Figure 5: The code of Feature in Isabelle/HOL.

```
datatype Connection-cate = PORT | PARAMETER |
DATA-ACCESS | SUBP-ACCESS

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram, 'Thread, 'Process, 'System) conn-conf =
cc-name :: string
cc-cate :: Connection-cate
cc-endp-src :: ('Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram, 'Thread, 'Process, 'System) Connection-ref
cc-endp-des :: ('Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram, 'Thread, 'Process, 'System) Connection-ref
cc-direction :: Connection-dir
cc-timing-type :: Timing-type option
cc-trans-type :: Transmission-type option
```

Figure 6: The code of Connection in Isabelle/HOL.

```
type_base ::=   [ type_name ]
                [ features { feature } + ]
```

Figure 7: The syntax of component type base.

```
subcomponent ::=   basic_type | data | subprogram
                   | thread | process | system
impl_base ::=      impl_name
                   [ connections { connection } + ]
                   [ subcomponents { subcomponent
                   } + ]
```

Figure 8: The syntax of component implementation base.

```
record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base =
type-features :: ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) Feature
set
type-name :: string
```

Figure 9: The code of component type base in Isabelle/HOL.

```
record ('Connection, 'Data, 'Subprogram, 'Thread, 'Process,
'System) impl-base =
impl-subcomps :: ('Data, 'Subprogram, 'Thread, 'Process, 'System)
Subcomponent set
impl-conns :: 'Connection set
impl-name :: string
```

Figure 10: The code of component implementation base in Isabelle/HOL.

4.1.3. Software Subcomponents. Software subcomponents represent the components contained within another software component. They can be the instantiations of component implementations if they are contained in their own subcomponents. As the statement of the core AADL selection in Subsection 3.1.1, the software subcomponents are specified as Data, Subprogram, Thread, Process, and System. Their instances are presented in Figure 11.

```
data_type :: =            type_base [data_properties]
data_impl :: =            impl_base [data_properties]
subprogram_type :: =      type_base [subpro-
                          gram_properties]
subprogram_impl :: =      impl_base [subpro-
                          gram_properties] [subprogram-
                          calls]
thread_type :: =          type_base [thread_properties] [be-
                          havior_annex]
thread_impl :: =          impl_base [thread_properties]
                          [subprogramcalls] [behav-
                          ior_annex]
process_type :: =         type_base [process_properties]
process_impl :: =         impl_base [process_properties]
system_type :: =          type_base [system_properties]
system_impl :: =          impl_base [system_properties]
```

Figure 11: The syntax of Subcomponents.

We consider that there is only one system as a parent component in a practically running model. This results in a component containment hierarchy that ultimately describes the whole actual system as a system instance. This section defines the following categories of software subcomponents: data subcomponent, subprogram subcomponent, thread subcomponent, process subcomponent, and system component. Their code snippets in Isabelle/HOL are presented as Figures 12–16.

4.1.4. Behavior Annex. As discussed in Subsection 3.1.2, a behavior specification subclause is a part of a thread, and it describes the thread that the Behavior annex belongs to. The Behavior annex is composed of variable set, state set, transition set, and its private information (like its name and ports of its parent component), and its elements are united by its transitions. A behavior_annex instance is presented in Figure 17.

The transitions can describe the behavior as a state-transition system linked with *guards* by some conditions and *actions*. The behavior_transition defines a relation from a source state to a destination state and represents a sequence of actions within a thread, which can be executed once its condition is satisfied. A behavior_transiton consists of its name, source state, destination state, guard condition, and actions. The actions associated with transitions are action block. A transition instance is presented in Figure 18.

The guards combined of conditions are in the transitions, which are explicitly classified as dispatch conditions and execution conditions. A dispatch condition is a Boolean expression that specifies the trigger of events. An execution condition is a logical expression on the inputs, outputs, values, and properties, or any other execute conditions. A transition instance is presented in Figure 19.

The actions can be classified as basic actions and action blocks in the transitions. The basic actions can be defined as empty (marked as NULL), assignment actions, communication actions, or timed actions. The action blocks are in the

```
record data-properties =
dt-access-right :: Access-right option
dt-concurrency-control-protocol :: Concurrency-Control-Protocol
option

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) data-type =
('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base +
dt-properties :: data-properties option

record ('Connection, 'Data, 'Subprogram, 'Thread, 'Process,
'System) data-impl = ('Connection, 'Data, 'Subprogram, 'Thread,
'Process, 'System) impl-base +
dt-properties :: data-properties option
```

Figure 12: The code of data in Isabelle/HOL.

```
record subprogram-properties =
sp-urgency :: int option
sp-compute-exetime :: TimeRange option
sp-compute-deadline :: Time option
sp-call-type :: Call-type option

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter)
subprogram-type =
('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base +
sp-properties :: subprogram-properties option

record ('Connection, 'Subprogramcalls, 'Data, 'Subprogram,
'Thread, 'Process, 'System) subprogram-impl =
('Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System)
impl-base +
sp-spcalls :: 'Subprogramcalls set
sp-properties :: subprogram-properties option
```

Figure 13: The code of subprogram in Isabelle/HOL.

form of sequences or sets. Every single action block is like imperative language and can be defined as conditionals and loops. Both assignment actions and communication actions consist of expressions; moreover, communication actions can reference for the events of initiating or freezing the parameters. Timed actions are a kind of predefining computation action. An action instance is presented in Figure 20.

The expressions consist of logical expressions, relational expressions, and arithmetic expressions. The values of expressions can be variables, constants, or the result of another expression, and the constants expression values can be Boolean, numeric or string literals, property constants, or property values. The presentation of this part is omitted as they are totally the same like the expression of imperative language.

According to all the related works above contributed to the formal specification, we define the syntax of the Behavior annex in Isabelle/HOL, and some parts are presented in Figure 21.

Notice: dispatcher is a predefined type, and it describes the hardware expression language used in the annex and is not given here; 's is a state set describing all possible values stored in ports; the action language of the annex is abstracted

```
record ('Port, 'Data, 'Subprogram, 'Thread, 'Process, 'System)
thread-properties = thd-stack-size :: Size option
thd-initialize-entrypoint :: ('Data, 'Subprogram, 'Thread, 'Process,
'System) Subcomponent option
thd-period :: Time option
thd-deadline :: Time option
thd-priority :: int option
thd-resumption-policy :: Resumption-policy list option
thd-deactivation-policy :: Deactivation-policy option
thd-dispatch-protocol :: Dispatch-protocol option
thd-dispatch-trigger :: 'Port list option
thd-dispatch-deadline :: Time option
thd-dispatch-offset :: Time option
thd-schedule-policy :: Schedule-policy option
thd-initialze-deadline :: Time option
thd-activate-deadline :: Time option
thd-deactivate-deadline :: Time option
thd-compute-deadline :: Time option
thd-recover-deadline :: Time option
thd-finalize-deadline :: Time option
thd-dispatch-time :: TimeRange option
thd-compute-time :: TimeRange option
thd-recover-time :: TimeRange option

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex)
thread-type = ('Port, 'Dataaccess, 'Subpaccess, 'Parameter)
type-base +
thd-properties :: ('Port, 'Data, 'Subprogram, 'Thread, 'Process,
'System) thread-properties option
thd-ba :: 'BehaviorAnnex option

record ('Port, 'Connection, 'Subprogramcalls, 'Data, 'Subprogram,
'Thread, 'Process, 'System, 'BehaviorAnnex) thread-impl =
('Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System)
impl-base +
thd-spcalls :: Subprogramcalls set
thd-properties :: ('Port, 'Data, 'Subprogram, 'Thread, 'Process,
'System) thread-properties option
thd-ba :: BehaviorAnnex option
```

Figure 14: The code of Thread in Isabelle/HOL.

as a function computing outputs from the value of its input ports. The time consumption of an action is directly modeled as a time attribute.

4.1.5. Whole Model. In our work, we aim at a running model for the next formal verification in a development. For this end, we define a whole system, which is described with a set of mapping between the datatypes and the configuration records in practice, and the whole system model syntax is presented in Figure 22.

4.2. Validation Rules for Grammar. AADL is a standard defined by the SAE, and its reversion was published in 2016. There are numerous methods and rules of description for AADL in the new version, and they cover syntax, naming rules, legality rules, consistency rules, and standard properties, and also several discrete and temporal semantics. However, none of current tools have integrated these methods and rules to check the AADL model comparatively at present, even they do not take these rules into account, especially the rules.

```
record process-properties = pro-period :: Time option
pro-priority :: int option
pro-resumption-policy :: Resumption-policy option
pro-deactivation-policy :: Deactivation-policy option
pro-load-exe-time :: TimeRange option
pro-load-deadline :: Time option
pro-startup-exe-time :: TimeRange option
pro-startup-deadline :: Time option

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) process-type =
('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base +
pro-properties :: process-properties option

record ('Connection, 'Data, 'Subprogram, 'Thread, 'Process,
System) process-impl = ('Connection, 'Data, 'Subprogram, 'Thread,
'Process, 'System) impl-base +
pro-properties :: process-properties option
```

Figure 15: The code of process in Isabelle/HOL.

```
record system-properties = sys-period :: Time option
sys-priority :: int option
sys-resumption-policy :: Resumption-policy option
sys-startup-deadline :: Time option

record ('Port, 'Dataaccess, 'Subpaccess, 'Parameter) system-type =
('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base +
sys-properties :: system-properties option

record ('Connection, 'Data, 'Subprogram, 'Thread, 'Process,
'System) system-impl = ('Connection, 'Data, 'Subprogram, 'Thread,
'Process, 'System) impl-base +
sys-properties :: system-properties option
```

Figure 16: The code of system in Isabelle/HOL.

```
behavior_annex ::=    [ variables { behavior_variable }+ ]
                      [ states { behavior_state }+ ]
                      [ transitions { behavior_transition }+ ]
```

Figure 17: The syntax of Behavior annex.

```
behavior_transition ::= [ trans_identifier [ [ trans_priority ] ] : ]
source_state_identifier { , source_state_identifier }+
                    –[ guard_condition ] →
    destination_state_identifier [ action_block ] ;
```

Figure 18: The syntax of Transition.

```
guard_condition ::= execute_condition
                  | dispatch_condition
execute_condition ::= logical_expression
                    | no_others
dispatch_condition ::= ondispatch [ trigger_condition ]
                       [ frozen ( frozen_ports ) ]
```

Figure 19: The syntax of guard.

```
action_block ::= "{" actions "}"
actions ::= action | action_sequence | action_set
action ::= basic_action
         | if ( logical_expression ) actions
             [ else actions ] end if
         | while ( logical_expression ) "{" actions "}"
         | for ( element_identifier in element_values )
             "{" actions "}"
basic_action ::= NULL
               | assignment_action
               | communication_action
               | timed_action
action_sequence ::= action { ; action }+
action_set ::= action{ & action }+
```

Figure 20: The syntax of Action.

Our work integrates the rules into the theorem prover Isabelle/HOL, which come to having a partial mapping from concrete syntax to abstract model. In this section, for the next formal verification built on the firmer trust basis, our work makes a link of the validation for the AADL model in Isabelle/HOL. Firstly, since some of all the rules are mandatory and others are recommended, our work determines 47 significant rules on account of AADL selection and they are considered to be compulsive. These rules cover software components, features, connections, and Behavior annex from the grammar perspective. Secondly, these rules R are specified as the definitions or functions by the functional language Isabelle/Isar in Isabelle/HOL as the properties needed to be validated. We specify the constraints as properties by using the function definition aim at guaranteeing the correctness of the built AADL model. And then, we abstract the element e of a realistic AADL model into Isabelle and instantiate all elements together into a concrete model M. It is mapped as a parameter of these rules definitions. Lastly, we identify the lemmas of these rule definitions and integrate into a comprehensive lemma about grammar. The correctness of the validation pass hinges on a lemma that shows the assertion:

$$M \vDash \text{grammar\_validate } (R\ e)$$

For the given lemma *grammar_validate*, the correctness of the validation pass is simple to state. Due to the space constraints, the segmental validation rules and definition code for grammar are classified as syntax, naming, and others (including legality, consistency, and stand properties), which are shown in Tables 2–4.

5. Formal Semantics and Verification

Formal semantics is a kind of mechanism based on strict and mathematical logic, which is especially important for describing safety-critical systems. Model's correctness and valid execution can be guaranteed by formal semantics. Although the AADL standard has depicted some execution semantics by natural language, it is the absence of precise dynamic semantics and even has no formal semantics at present. In addition, the AADL model cannot be executed directly because it is just an abstract description of the

```
datatype Behavior-state-kind = INITIAL | COMPLETE | FINAL |
EXECUTION
type-synonym BA-state = string × ( Behavior-state-kind set )
type-synonym 's bexp = 's set

datatype ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram) BA-action = Skip
| Basic-Assign 's ⇒ 's
| Basic-CommunSend ('Port, 'Dataaccess, 'Subpaccess, 'Parameter)
Feature 's ⇒ Data Message
| Basic-CommunRecv ('Port, 'Dataaccess, 'Subpaccess, 'Parameter)
Feature 'Data Message ⇒ 's
| Basic-CommunFreeze
| Basic-CommunInisend
| Basic-CommunCallsp 'Subprogram 's ⇒ 's
| Seqs ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram) BA-action ('s, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram) BA-action
| Sets ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram) BA-action ('s, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram) BA-action
| If 's bexp ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) BA-action ('s, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram) BA-action
| While 's bexp ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) BA-action

datatype ('s, 'Dispatcher, 'Port, 'Subpaccess) Behavior-Condition =
DispatchCond 'Dispatcher
| DispatchCond-TriggerLogicExp ('Dispatcher option × 'Port) set
| DispatchCond-Subpaccess 'Dispatcher option × 'Subpaccess
| DispatchCond-Stop 'Dispatcher × Event
| DispatchCond-Timeout 'Dispatcher × Time option
| ExecuteCond-LogicExp 's bexp
| ExecuteCond-Timeout Time option

record ('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) BA-transition = src-state :: BA-state
des-state :: BA-state
condition :: ('s, 'Dispatcher, 'Port, 'Subpaccess) Behavior-Condition
option
actions :: ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Data,
'Subprogram) BA-action list

record ('Data, 'Subprogram, 'Thread, 'Process, 'System) BA-var =
var-name :: string
```

Figure 21: The code of Behavior annex in Isabelle/HOL.

system architecture. This not only restricts the possibility of formal analysis of AADL models, but cannot conduct model checking and verification. It is necessary to propose a way to specify AADL models with formal models. However, AADL is mainly a mathematical model, which cannot be used to automatically verify properties of a given AADL model. In order to provide evidence of model checking and enable the proof of semantics preservation of system running, the informal execution semantics of AADL formalized directly in Isabelle/HOL are considered an operational semantics.

In this article, the AADL semantics is given an operational semantics, which is delivered from the AADL

```
AADL_model ::=  { features }
                { connections }
                { subprogramcalls }
                { data_type }
                { data_impl }
                { subprogram_type }
                { subprogram_impl }
                { thread_type }
                { thread_impl }
                { process_type }
                { process_impl }
                { system_type }+
                { system_impl }
                { behavior_annex }
```

Figure 22: The syntax of the model.

informal execution semantics and can be compared with the informal one. The main benefit with operational semantics is that it is based on a rigorous mathematical foundation and is built on the same principles as functional programming languages. Such benefits determined us to define an underlying operational semantics for AADL and its Behavior Annex, and consequently implement the verification in Isabelle/HOL.

5.1. Semantics of Behavior Annex. The AADL model is completed with behavioral descriptions using the Behavior annex, like computation and communication for threads. Thus, there is a relation between the AADL execution model and the Behavior annex. The execution model specifies when the Behavior annex is executed and on which data it is executed, while the Behavior annex acts in a thread (or a subprogram) and describes behaviors more precisely. For this purpose, the semantic specifications given as above will be enriched by the Behavior annex. Since the behavior is explicitly expressed by atomic transitions, the operational semantics of the Behavior annex is defined based on the refinement of the rule of each transition in the Behavior annex, and the execution semantics of the Transition is based on the semantics of the Actions in itself. This section begins by describing how to formalize the meaning of Behavior annex using automaton. And then, the constituents of the Behavior annex and their semantics are defined (including transition system, action and expression language, etc.).

5.1.1. Formalization of Behavior Annex by Automaton. We present the formalization of a Behavior annex by using an incomplete automaton AM with several variables. The AM is used to interpret the meaning of the whole Behavior annex, and AM = (S, $s0$, V, P, B, T, C) is defined as

(i) S: the states set of AM.

(ii) $s0$: the initial state, $s0 \in S$.

(iii) V: the local variables set of AM.

(iv) P: the ports set of AM (including the input set IP and the output set OP, $P = \text{IP} \cup \text{OP}$).

Table 2: Description and code of syntax rules.

Description	Code in Isabelle/HOL
(N1) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.	definition type_name_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "type_name_valid m ≡ (∀dt1 dt2. (data_tp m)≠None ∧ dt1≠dt2 ⟶ (type_name (the (data_tp m) dt1))≠(type_name (the (data_tp m) dt2))) ∧ (∀sp1 sp2. (subprogram_tp m)≠None ∧ sp1≠sp2 ⟶ (type_name (the (subprogram_tp m) sp1))≠(type_name (the (subprogram_tp m) sp2))) ∧ (∀thd1 thd2. (thread_tp m)≠None ∧ thd1≠thd2 ⟶ (type_name (the (thread_tp m) thd1))≠(type_name (the (thread_tp m) thd2))) ∧ (∀pro1 pro2. (process_tp m)≠None ∧ pro1≠pro2⟶ (type_name (the (process_tp m) pro1))≠(type_name (the (process_tp m) pro2))) ∧ (∀sys1 sys2. sys1≠sys2 ⟶ (type_name ((system_tp m) sys1))≠(type_name ((system_tp m) sys2)))"
(N1) A component implementation name consists of a component type identifier and a component implementation identifier separated by a dot (“.”). The first identifier of the defining component implementation name must name a component type that is declared in the same package as the component implementation, or name an alias to a component type in another package.	definition impl2type_name_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "impl2type_name_valid m ≡ (∀dt. (data_im m)≠None ∧ (data_tp m)≠None ∧ (the (data_im m) dt)≠None ⟶ (get_prename_impl_dt m dt)=(type_name (the (data_tp m) dt))) ∧ (∀sp. (subprogram_im m)≠None ∧ (subprogram_tp m)≠None ∧ (the (subprogram_im m) sp)≠None ⟶ (get_prename_impl_sp m sp)=(type_name (the (subprogram_tp m) sp))) ∧ (∀thd. (thread_im m)≠None ∧ (thread_tp m)≠None ∧ (the (thread_im m) thd)≠None ⟶ (get_prename_impl_thd m thd)=(type_name (the (thread_tp m) thd))) ∧ (∀pro. (process_im m)≠None ∧ (process_tp m)≠None ∧ (the (process_im m) pro)≠None ⟶ (get_prename_impl_pro m pro)=(type_name (the (process_tp m) pro))) ∧ (∀sys. (system_im m)≠None ∧ (the (system_im m) sys)≠None ⟶ (get_prename_impl_sys m sys)=(type_name ((system_tp m) sys)))"
.....	. . .. . .

(v) B: the Boolean formulas set of AM (these multisorted logical formulas are defined over the vocabulary available in the lexical scope of a Behavior annex: AADL value constants, port, state, and variable names, $B = S \cup V \cup P$).

(vi) T: the transition function set defines the transition system of AM, $T \in F \times S \longrightarrow F \times S$. Each specified transition has its quadruple (s, g, a, d) which defines the source state s, guard formula g, action formula a, and destination state t. includes the guard g and the action a.

(vii) g denotes the source formula of a transition defined on V and I.

(viii) a represents the target formula of a transition defined from V and P.

(ix) C: the constraint set of AM, which denotes the invariants (properties, requirements) of the denoted AADL object and denoted by the multisorted logical formulas, $C \in B$. It must always equal 0.

If the thread is in dispatch status, the states (initial, complete, final) of Behavior annex can be observed, as these states are specified in a transition and they can be mapped to the execution of the outside components. If the thread is under execution, the detail of states cannot be observed, as their execution states are just held in a running thread and they are the internal states related to the processor in the whole Behavior annex. So our work defines the formal execution semantics of Behavior annex as two parts: one is about transitions, and the other is about Actions in transitions. These two parts are described by big-step semantics through a global state (shared variables), and they can be recombined to the integrated semantics of whole Behavior annex. Furthermore, the Action semantics can be mapped to some imperative language in the future work, and its formal semantics would make the preservation more clear.

Table 3: Description and code of naming rules.

Description	Code in Isabelle/HOL
(N1) The defining identifier for a component type must be unique in the namespace of the package within which it is declared.c	definition type_name_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "type_name_valid m ≡ (∀dt1 dt2. (data_tp m)≠None ∧ dt1≠dt2 ⟶ (type_name (the (data_tp m) dt1))≠(type_name (the (data_tp m) dt2))) ∧ (∀sp1 sp2. (subprogram_tp m)≠None ∧ sp1≠sp2 ⟶ (type_name (the (subprogram_tp m) sp1))≠(type_name (the (subprogram_tp m) sp2))) ∧ (∀thd1 thd2. (thread_tp m)≠None ∧ thd1≠thd2 ⟶ (type_name (the (thread_tp m) thd1))≠(type_name (the (thread_tp m) thd2))) ∧ (∀pro1 pro2. (process_tp m)≠None ∧ pro1≠pro2⟶ (type_name (the (process_tp m) pro1))≠(type_name (the (process_tp m) pro2))) ∧ (∀sys1 sys2. sys1≠sys2 ⟶ (type_name ((system_tp m) sys1))≠(type_name ((system_tp m) sys2)))"
(N1) A component implementation name consists of a component type identifier and a component implementation identifier separated by a dot (".“). The first identifier of the defining component implementation name must name a component type that is declared in the same package as the component implementation, or name an alias to a component type in another package.	definition impl2type_name_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "impl2type_name_valid m ≡ (∀dt. (data_im m)≠None ∧ (data_tp m)≠None ∧ (the (data_im m) dt)≠None ⟶ (get_prename_impl_dt m dt)=(type_name (the (data_tp m) dt))) ∧ (∀sp. (subprogram_im m)≠None ∧ (subprogram_tp m)≠None ∧ (the (subprogram_im m) sp)≠None ⟶ (get_prename_impl_sp m sp)=(type_name (the (subprogram_tp m) sp))) ∧ (∀thd. (thread_im m)≠None ∧ (thread_tp m)≠None ∧ (the (thread_im m) thd)≠None ⟶ (get_prename_impl_thd m thd)=(type_name (the (thread_tp m) thd))) ∧ (∀pro. (process_im m)≠None ∧ (process_tp m)≠None ∧ (the (process_im m) pro)≠None ⟶ (get_prename_impl_pro m pro)=(type_name (the (process_tp m) pro))) ∧ (∀sys. (system_im m)≠None ∧ (the (system_im m) sys)≠None ⟶ (get_prename_impl_sys m sys)=(type_name ((system_tp m) sys)))"
...	...

5.1.2. Semantics of Transition in Behavior Annex. Actually, the transition system, which is described by three sections (variable declarations, state declarations, and transition declarations mentioned in Section 4.1.4), is a refinement of the AADL Behavior annex, and it is created by linking two states using a guarded automaton. A transition (Si − [conditions] ⟶ Sj{actions}) specifies the behavior as a state change from a source state Si to a destination state Sj, which can be guarded by conditions (dispatch or execution). In this section, we specify the meaning of the elements in transition and use them to express the semantics of transition.

(1) Variables: The variables, which are temporary through the whole Behavior annex subclause, declare the identifiers that represent local variables and record intermediate results within the scope of the whole Behavior annex subclause. They can be used to hold the values of out parameters on subprogram calls and also can hold input from incoming port queues or values read from data components in the AADL specification.

(2) States: The states, which may be mapped to the various thread states, are categorized as initial, complete, final, or execution state. The initial state means thread state halted, the complete state means thread state awaiting for dispatch (suspend or resume), the final state means thread state stopped, and the execution state means the rest of thread states (running) that are not be observable.

(3) Transitions: The transitions define an execution automaton in a thread. They may be guarded by dispatch or execute conditions, and the sequence of actions within each transition can be executed atomically when their conditions are satisfied. Dispatch conditions explicitly specify dispatch trigger conditions out of a complete state to another state. Execute conditions specify transition conditions out of an execution state. Actions can be subprogram

Table 4: Description and code of other rules.

Description	Code in Isabelle/HOL
(1) A thread models a concurrent task or an active object, that is, a schedulable unit that can execute concurrently with other threads. Each thread represents a sequential flow of control that executes instructions within a binary image produced from the source text. One or more AADL threads may be implemented in a single operating system thread. A thread always executes within the virtual address space of a process; that is, the binary images making up the virtual address space must be loaded before any thread can execute in that virtual address space. Threads are dispatched; that is, their execution is initiated periodically by the clock or by the arrival of data or events on ports, or by the arrival of subprogram calls from other threads.	definition thread_thread2system_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "thread_thread2system_valid m ≡ ∀sys. ∃sc1 sc2. (if ((system_im m)≠None ∧ (the (system_im m) sys)≠None ∧ (impl_subcomps (get_sysimpl m sys))≠{}) then (sc1∈(impl_subcomps (get_sysimpl m sys)) ⟶ (case sc1 of SCThd _ ⇒ True | SCPro pro ⇒ (if ((process_im m)≠None ∧ (the (process_im m) pro)≠None ∧ (impl_subcomps (get_proimpl m pro))≠{}) then (sc2∈(impl_subcomps (get_proimpl m pro)) ⟶ (case sc2 of SCThd _ ⇒ True | _ ⇒ False)) else False) | _ ⇒ False)) else False)" definition thread_thread2process_valid:: "('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter, 'Subprogramcall, 'Subprogramcalls, 'Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System, 'BehaviorAnnex) AADL_model ⇒ bool" where "thread_thread2process_valid m ≡ ∀pro. ∃sc. (if ((process_im m)≠None ∧ (the (process_im m) pro)≠None ∧ (impl_subcomps (get_proimpl m pro))≠{}) then (sc∈(impl_subcomps (get_proimpl m pro)) ⟶ (case sc of SCThd _ ⇒ True | _ ⇒ False)) else False)"
...	...

calls, receiving of input and sending of output, assignments to variables, read/write to data components, or other activities.

We describe a transition as a series of atomic operation in Behavior annex, and a series of Transitions are composed into a whole behavior. As a transition is mainly supported to complete the thread component with behavior handling inputs and outputs in order to enrich the communication mechanism, so in some sense, talking about the semantics of whole transitions is about Behavior annex actually. Our work defines the semantics of Behavior annex by the operational big-step semantics presented by the automaton AM as T=(s, g, a, d), and they are defined inductively as follows:

T: 's BA_Transition

s, d: State

a: Actions

g: Conditions

The core state space is denoted in terms of states by the type 's different from the state of transitions, it is polymorphic, and its semantics is augmented with control flow information in Isabelle/HOL as follows.

Where the type variable state 's is regarded as a set of variable states and related to the variable state in Actions when the execution is in a normal state Normal s. Besides, 's can hold the value of variables on inputs or outputs, and it is attached to the receiving or sending events as a message Msg s as an option.

Moreover, a transition is guarded by conditions, and conditions can be either dispatch conditions or execution conditions as shown in Figure 19. We consider a dispatch condition g of T formed by clock c as its dispatch trigger; it means the dispatch condition is presented as time trigger. An execution condition g of T is considered logical value expressions. The semantic of transition is defined inductively by the rules in Isabelle/HOL as shown in Figure 23.

5.1.3. Semantics of Action in Behavior Annex. This section defines the semantics of Actions by the operational big-step semantics. Actions of the Behavior annex define actions performed during transitions. Actions associated with transitions are action blocks that are presented in Figure 20, where the single action can be defined as control structures such as basics, conditionals, and loops. The action_sequence means it is executed in order, while action_set can be executed in any order.

The basic_action can be empty, assignment, communication, or time-consuming. The assignment_action is reference for the value assignment. The communication_action is reference for receiving and sending data, event, or event data on the inputs and outputs. The Timed_action is predefined computation actions.

Since this section should be related to the semantics of transitions and be a part of it, the semantics of Action is defined as $\Gamma: : \langle s, a\rangle \Rightarrow d$, which means in the procedure environment Γ execution transforms the source state s to the destination state d. Actually, the destination state d is the

```
type-synonym ('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram, 'Thread, 'Process, 'System,
BehaviorAnnex) BA-body
                    = 'BehaviorAnnex ⇒ ('s, 'Dispatcher, 'Port,
'Dataaccess, 'Subpaccess, 'Parameter, 'Data, 'Subprogram, 'Thread,
'Process, 'System) behavior-annex-conf option

inductive
  BA-bigstep :: [('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram, 'Thread, 'Process, 'System,
'BehaviorAnnex) BA-body,
                  ('s, 'Dispatcher, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram) BA-transition,
                    ('s, 'p, 'f, 'Dispatcher, 'Port, 'Dataaccess,
'Subpaccess, 'Parameter, 'Data, 'Subprogram, 'Thread, 'Process,
'System, 'BehaviorAnnex) AADL-state,
                    ('s, 'p, 'f, 'Dispatcher, 'Port, 'Dataaccess,
'Subpaccess, 'Parameter, 'Data, 'Subprogram, 'Thread, 'Process,
'System, 'BehaviorAnnex) AADL-state] ⇒ bool
          (-⊢⟨-,-⟩→- [93,92,94,94] 95)
                      for BA-BODY :: ('s, 'Dispatcher, 'Port,
'Dataaccess, 'Subpaccess, 'Parameter, 'Data, 'Subprogram, 'Thread,
'Process, 'System, 'BehaviorAnnex) BA-body
where
Tran-ini2com: [[BA-BODY BA = Some ba-conf;
          ba-tran ∈ (ba-trans ba-conf);
          ba-var ∈ (ba-vars ba-conf);
          (snd (src-state ba-tran)) = INITIAL;
          (snd (des-state ba-tran)) = COMPLETE;
        (snd ((ba-st s) BA)) = INITIAL ∧ (fst (src-state ba-tran))
= (fst ((ba-st s) BA));
          (snd ((ba-st t) BA)) = COMPLETE ∧ (fst (des-state
ba-tran)) = (fst ((ba-stt) BA));
          (condition ba-tran) = None
          ∨ (the (condition ba-tran) = DispatchCond -)
        ∨ (the (condition ba-tran) = DispatchCond-TriggerLogicExp
{(-,-)})
          ∨ (the (condition ba-tran) = DispatchCond-Subpaccess
(-,-))
          ∨ (the (condition ba-tran) = DispatchCond-Stop (-,-))
          ∨ (the (condition ba-tran) = DispatchCond-Timeout (-,-))
          ∨ ((the (condition ba-tran) = ExecuteCond-LogicExp be)
∧ (case s1 of Normal s' ⇒ s' ∈ be));
                    acts=(actions-block ba-tran);
          s1=((ba-var-st s) ba-var);
          s2=((ba-var-st t) ba-var);
          ACT-BODY⊢⟨acts,s1⟩ ⇒ s2]]
          ==⇒BA-BODY⊢⟨ba-tran, s⟩ →t
.....
```

Figure 23: Big-step semantics of transition in Isabelle/HOL.

destination state d of its transition. To connect with the automaton T, we refinement a in the automaton T as a new automaton A and reconstitute the automata T'=(s, g, true, t) ∪ A and A=(t, true, a, d) where a in A and a in the previous T are not same, and the former is the detailed implementation of the latter. T′ and A are defined by the case on Actions as follows:

(i) Action sequence is a list type of executions. For example, action_sequence a = [a1; a2] separates A=(t, true, a, d) into A1=(t, true, a1, t') and A2=(t', true, a2, d) by introducing an intermediate state t', and then, T'=(s, g, true, t) ∪ A1 ∪ A2 by the union of them.

(ii) Action set is a disorderly combination of executions. For instance, action_set a = [a1 & a2] makes that A =(A1 × A2), which the composed elements are A1 = (t1, true, a1, d1), A2 = (t2, true, a2, d2), t=(t1, t2), and d=(d1, d2).

(iii) Empty_action of basic_action NULL can be represented as an invalid operation to the states and defined by SKIP.

(iv) Assignment_action of basic_action v\coloneq e represents a variable state transition and it is defined by A = (t, true, v' = e, d) where v' represents the successor state of v.

(v) Communication_action of basic_action is divided into the output port action port!(*e*) and the input port action port?(v). The output can be defined by A=(t, true, port = e, d). The input can be defined by A=(t, true, v' = port, d) where v' represents the successor state of v.

(vi) Timed_action of basic_action computation(t1. . .tN) represents the execution time of the action block. It is specified as two ports—pb (port begin) and pe (port end), and defined by A = {(s, true, pb, i), (i, pe, true, d)}, where i is an intermediate state as a complete state and timed constraint Val(pb + t1) ≤ Val(pe + tN).

(vii) Conditional_action, for instance, if(exp) a1 else a2 end if can be defined by T' = {(s, g, true, t1), (s, g, true, t2)} ∪ A1 ∪ A2 which A1 = (t1, true, a1, d), A2 = (t2, true, a2, d), and the guard g is corresponding to the logical expression exp in conditional_action.

(viii) While_loop_action, for instance, while(exp) { a } can be defined by T' = {(s, g, t g, true, d), (t2, g, true, t1), (t2, g, true, d)} ∪ A, where the guard g is corresponding to the logical expression exp in while_loop_action and A=(t1, true, a, t2).

(ix) For_loop_action, for instance, for(I in *e*) { a } can be translated by the action sequence [a1; . . .; an], where ai results from the substitution of i by the i_th element value of e in a.

The semantic of Action is defined inductively by the rules in Isabelle/HOL as shown in Figure 24. The procedure environment *act-body* denotes the static procedure declarations as mapping from subprogram names to actions programs of Behavior annex and defines the execution of command c that transforms the initial state s to the final state t under *act-body*.

5.1.4. Semantics of Expressions in Behavior Annex. Expressions consist of logical expressions, relational expressions, and arithmetic expressions like the expressions of imperative language. Values of expressions can be variables, constants, or the result of another expression. In AADL, expressions are used as logical conditions of guards in transitions or logical expressions in conditional actions, or as values for basic actions. Values of variable expression are evaluated from inputs, local variables, and data subcomponents. Values of constant expression are Boolean, numeric or string literals, and property values. In our work, expressions are defined by the type variables's as a set of states in Isabelle/HOL and related to the variable state in Actions.

5.2. Semantics of AADL Components. There are a great number of components that can be used to build hierarchical models in AADL, and it makes AADL have a great capacity of expression. In our work, the AADL model is viewed as a set of concurrent tasks scheduled by a processor and asynchronously interacted. Generally, we consider the following components: data, thread, process, and processor. These components are connected through AADL port connections, completed with a set of standard properties, and finally grouped in the system component. However, in a running system model, a process component represents the virtual address space and scheduled by the processor. Indeed, a thread component is the minimum schedulable unit under execution, and then, they are concepts that require detailed attention as they include the behavior of AADL. What's more, our goal is to verify system behavior, so in this section, we focus our experimentation on software components in the software model and thread management. Besides, the mode semantics is not yet completely stabilized in the standard so we take no account of mode management, and our work highlights several constructions(like Global_Timer, Dispatcher, and Scheduler) to make the AADL system model running and also its semantics of components more coherent.

According to the AADL standard, the running model of software components can be described by execution automaton; the following paragraph describes that the software components are applied to the execution automaton and the management of communication (Figure 25).

5.2.1. Semantics of Thread Component Execution Model. First of all, the necessary Thread execution model elements are currently specified according to the AADL running models by two points—dispatching and scheduling, and they both can be expressed by an automaton as shown in Figure 26.

```
type-synonym  ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) act-body = 'Subprogram ⇒ ('s, 'Port,
'Dataaccess, 'Subpaccess, 'Parameter, 'Data, 'Subprogram)
BA-action option
inductive
BA-action-bigstep :: [('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) act-body,
                      ('s, 'Port, 'Dataaccess, 'Subpaccess, 'Parameter,
'Data, 'Subprogram) BA-action, 's vstate, 's vstate] ⇒ bool
                  (-⊢⟨-,-⟩ ⇒ -[97,96,98,98] 99)
              for ACT-BODY :: ('s, 'Port, 'Dataaccess, 'Subpaccess,
'Parameter, 'Data, 'Subprogram) act-body
where
SKIP: ACT-BODY⊢⟨Skip, Normal s⟩ ⇒ Normal s
| Assign: ACT-BODY⊢⟨Basic-Assign ba, Normal s⟩ ⇒ Normal (ba
s)
| CommunSend: ACT-BODY⊢⟨Basic-CommunSend bcs, Msg m⟩ ⇒
Msg (bcs m)
| CommunRecv: ACT-BODY⊢⟨Basic-CommunRecv bcr, Normal s⟩ ⇒
Normal (bcr s)
| CommunFreeze: ACT-BODY⊢⟨Basic-CommunFreeze, Normal s⟩ ⇒
Normal s
| CommunInisend: ACT-BODY⊢⟨Basic-CommunInisend, Normal s⟩ ⇒
Normal s
| CommunCallsp: [[ACT-BODY sp=Some baact;
          ACT-BODY⊢⟨Basic-CommunCallsp sp paras, Normal s⟩ ⇒
Normal (paras s);
           ACT-BODY⊢⟨baact,Normal (paras s)⟩⇒t]]
        ==⇒ ACT-BODY⊢⟨Basic-CommunCallsp sp paras, Normal
s⟩ ⇒ t
| Seqs: [[ACT-BODY⊢⟨a1, Normal s⟩⇒s';
     ACT-BODY⊢⟨a2, s'⟩ ⇒ t]]
   ==⇒ ACT-BODY⊢⟨Seqs a1 a2, Normal s⟩ ⇒ t
| Sets: [[ACT-BODY⊢⟨a1, Normal s⟩ ⇒ t;
     ACT-BODY⊢⟨a2, Normal s⟩ ⇒ t]]
   ==⇒ ACT-BODY⊢⟨Sets a1 a2, Normal s⟩ ⇒ t
| IfTrue: [[s∈be; ACT-BODY⊢⟨a1, Normal s⟩ ⇒ t]]
      ==⇒ ACT-BODY⊢⟨If be a1 a2, Normal s⟩ ⇒ t
| IfFalse: [[s∉be; ACT-BODY⊢⟨a2, Normal s⟩ ⇒ t]]
        ==⇒ ACT-BODY⊢⟨If be a1 a2, Normals⟩⇒t
| WhileTrue: [[s∈be;
          ACT-BODY⊢⟨a, Normal s⟩ ⇒ s';
          ACT-BODY⊢⟨While be a, s'⟩ ⇒ t]]
        ==⇒ACT-BODY⊢⟨While be a, Normal s⟩ ⇒ t
| WhileFalse: [[s∉be]]
        ==⇒ ACT-BODY⊢⟨While be a, Normal s⟩ ⇒ Normal s
```

Figure 24: Big-step semantics of Action in Isabelle/HOL.

The dashed box represents Thread_Computation state, and in the Thread_Computation of the execution automaton, the Thread internal behavior is carried out according to the behavior expression, which was described in Section 5.1. In order to integrate all the Thread execution states in one Thread, the other execution states are generated for each Thread. Every Thread execution states transition is managed by its previous states, its parent Process, and System execution state. They also are both decided by the next state of the Transition in Behavior annex, so the semantics of Transition are embedded in this part for connecting with Thread_Computation. In the case of execution Initialize, the

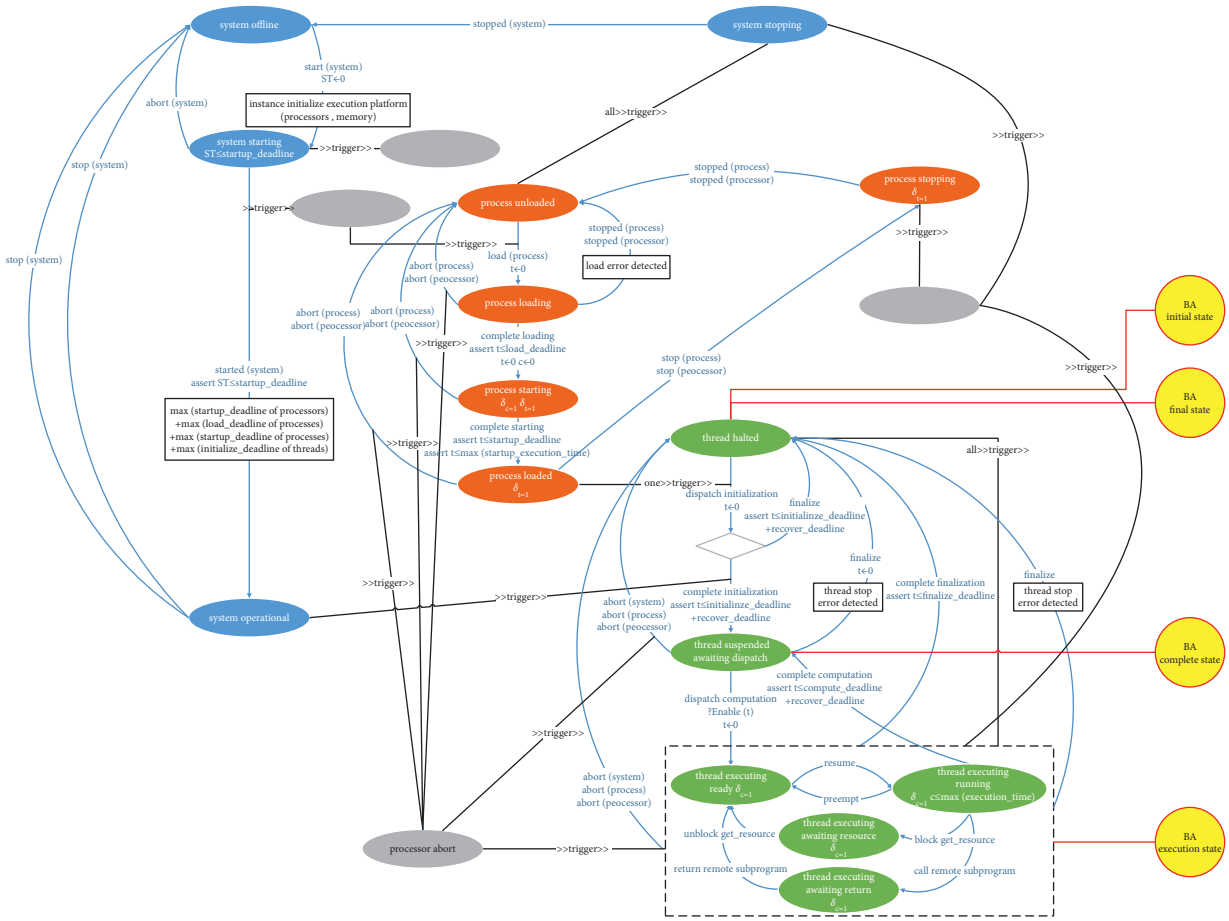

Figure 25: Whole execution automaton.

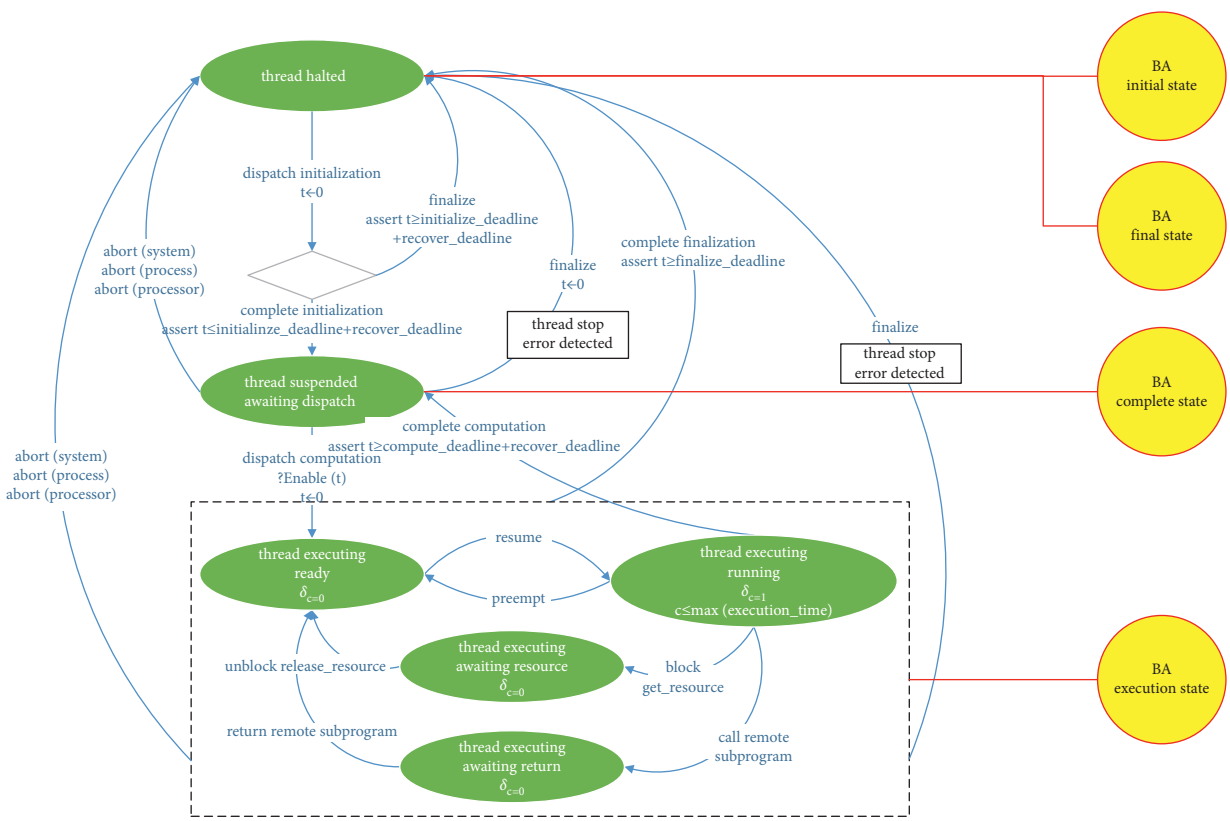

Figure 26: Thread execution automaton.

code of state conditions management part in Isabelle/HOL is shown in Figure 27.

In the transitions automaton, the execution time, the elapsed time, and waiting time are controlled by the creation of the global clock and the various Deadline (initialize_Deadline, compute_Deadline, recover_Deadline, etc.), and they provide the possibility to manage several kinds of Thread including periodical, aperiodic, and sporadic ones. In the case of execution Initialize, the code of temporal part in Isabelle/HOL is shown in Figure 28.

To support sending and receiving messages (data, event, and data_event) between components, the Connections provide the communication mechanism to manage messages from the source to the destination point. They are typed with Access_Connections and Port_Connections. The Access_Connections type is used to model the data flow shared by access between components like Subprogram_Access and Data_Access, and the Port_Connections type is used to model transfer of data or events between ports. All types also include the Parameter_Connections, which models a data flow representing the parameter of subprogram included in a Thread, but this type is not managed in our transformation on account of the practical frequency.

Now, we focus our presentation on the Port_Connection type. It deals with the processing of the sent and received messages and the properties describe several behavioral features (like the Queue_Size, Queue_Processing_Protocol, and some other properties) to define a queue of messages associated with a port. The processing of the messages received by Thread is carried out when it is in the execution state Thread_Computation. This state is reached after the dispatch of Thread. Unfortunately, these necessary concepts are not described explicitly in the AADL standard, so we should take into account an execution model definition.

In our work, several processing conditions are added on the Thread semantic to specify the Connection model. For example, the port queue state is estimated and the messages on the queue are handled by a dispatch mechanism. This mechanism is dedicated to detach the internal behavior of a Thread and the message consumption. In the Thread execution model, the dispatch action is performed on the transition between the state Thread_Suspend_Waiting_Dispatch and Thread_Computation. Generally, the message once arrives on the ports will be copied in variables in the Behavior of the Thread through the dispatch. In the execution state of Thread_Computation, the Thread handles its behavior with data and event copies. So, we combine the state of the Behavior and add the definitions "is_port_queue_empty" and "handle_port" to specify and to conceptualize the Connections based on the AADL standard properties. In the case of execution Initialize, the code in Isabelle/HOL is presented in Figure 29. For these additions, they are mainly used at a design level for code generation and they also make the semantics of Behavior add into the model execution semantics to be a whole complete and continuous semantics.

5.2.2. Semantics of Process Component Execution Model. A process represents a virtual address space at runtime, so the Process execution model is driven by the processor mainly and it works on the state to affect its own Thread execution model inside in effect. We consider that the Process execution model is managed by the clock and express its parent System execution state as an automaton as shown in Figure 30.

In the transitions automaton, the execution time, the elapsed time, and waiting time are controlled by the creation of the global clock and the various Deadline (load_Deadline, startup_Deadline, etc.), and also the prestate and poststate must be satisfied. In the case of execution Load, the code of the process execution Load part in Isabelle/HOL is shown in Figure 31.

5.2.3. Semantics of System Component Execution Model. A system represents the runtime architecture of an actual system that consists of application software components and execution platform components, and it is the top hierarchy of the whole execution model, so the System execution model is only driven by the processor and it works on the state to affect its own Process and Thread execution model inside in effect. Same as the Process execution model, the system execution model is managed by the clock and it is presented as an automaton as shown in Figure 32.

In the case of execution Start_Complete, the code of the System execution Start_Complete part in Isabelle/HOL is shown in Figure 33.

5.3. Formal Verification for AADL

5.3.1. Proof system Framework. The calculation of the AADL execution model is actually a sequence of transitions. The computations set for whole executions with static information Σ is defined as $\Gamma(\Sigma)$. We use function $\Gamma(\Sigma, \rho, s, e)$ to present the computations of an execution system ρ starting up from an initial state s and execution e. A configuration of computation is defined as a triple $\delta = (\theta, s, e)$ where θ is specified as transition rules in execution model systems, which have the form $\Sigma \vdash (\theta_n, s_n, e_n) \longrightarrow (\theta_{n+1}, s_{n+1}, e_{n+1})$.

A specification in the proof system is a tuple $\langle pre, pst \rangle$, where *pre* is short for the precondition, and *pst* stands for the postcondition. For each computation $\delta \in \Gamma(\Sigma, \rho, s, e)$, the configuration at index i is denoted by δ_i, and we use θ_{δ_i}, s_{δ_i} and e_{δ_i} to signify the element inside $\delta_i = (\theta, s, e)$. We use A and C to denote assumption and commitment functions, respectively.

```
process ∈ (get-pros-bysys m system);
thread ∈ (get-thds-bypro m process);
port ∈ (get-ports-bythd m thread);
subpaccess ∈ (get-subpaccesses-bythd m thread);
dataaccess ∈ (get-dataaccesses-bythd m thread);
ps1=((port-st s) port) ∧ spaccs1=((spacc-st s) subpaccess);
(pr-state ((pr-st s) process))=PRO-LOADED ∧ (pr-state ((pr-st t)
process))=PRO-LOADED;
(th-state ((th-st s) thread))=HALTED;
(th-state ((th-st t) thread))=SUSPENDED-WAITING-DISPATCH;
(snd ((ba-st s) th-ba))=INITIAL ∧ fst ((ba-st s) th-ba) = fst
(src-state th-ba-tran);
(snd ((ba-st s) th-ba))=INITIAL ∧ fst ((ba-st s) th-ba) = fst
(src-state th-ba-tran);
(snd ((ba-st t) th-ba))=COMPLETE ∧ fst ((ba-st t) th-ba) = fst
(des-state th-ba-tran);
if ((is-port-queue-empty s port)=False)
    then ((ps2=handle-port s port) ∧ (spaccs2=spaccs1))
    else ((spaccs2=handle-spaccess s subpaccess) ∧ (ps2=ps1));
((port-st t) port) = ps2 ∧ ((spacc-st t) subpaccess) = spaccs2;
(BA-BODY th-ba) = Some th-ba-conf;
BA-BODY⊢⟨th-ba-tran, s⟩ → t
```

Figure 27: The state conditions management of Thread execution Initialize semantic.

```
gt = (cur-time s);
(begin-time ((th-st s) thread)) = (cur-time s);
(action-begin-time ((th-st s) thread)) = (cur-time s);
(elapsed-time ((th-st t) thread)) ≤ (the (thd-initialze-deadline
(the (thread-type.thd-properties (the (thread-tp m) thread)))))+(the
(thd-recover-deadline (the (thread-type.thd-properties (the (thread-tp
m) thread)))));
(cur-time t) = (cur-time s) + (elapsed-time ((th-st t) thread));
(cur-time t) = (action-begin-time ((th-st s) thread)) + (elapsed-time
((th-st t) thread));
(exe-time ((th-st t) thread)) ≤ (elapsed-time ((th-st t) thread));
```

Figure 28: The temporal part of Thread execution Initialize semantic.

```
if ((is-port-queue-empty s port)=False)
then ((ps2=handle-port s port) ∧ (spaccs2=spaccs1))
else ((spaccs2=handle-spaccess s subpaccess) ∧ (ps2=ps1));
((port-st t) port) = ps2 ∧ ((spacc-st t) subpaccess) = spaccs2;
```

Figure 29: The connection part of Thread execution Initialize semantic.

$$A(\Sigma, pre) \equiv \left\{\rho \mid s_{\delta_i} \in pre \wedge \left(\forall i < (\text{length } (\delta) - 1) \cdot \left(\Sigma \vdash \delta_i \longrightarrow \delta_{i+1}\right) \longrightarrow \left(s_{\delta_i}, s_{\delta_{i+1}}\right)\right)\right\},$$
$$C(\Sigma, pst) \equiv \left\{\rho \mid \left(\forall i < (\text{length}\,(\delta) - 1) \cdot \left(\Sigma \vdash \delta_i \longrightarrow \delta_{i+1}\right) \longrightarrow \left(s_{\delta_i}, s_{\delta_{i+1}}\right)\right) \wedge \left(\theta_{\text{last}\,(\delta)} = \lfloor \perp \rfloor \longrightarrow s_{\delta_n} \in pst\right)\right\}. \tag{1}$$

We define validity of specification for executions as

$$\Sigma \vDash \rho \text{ sat } \langle \text{pre}, \text{pst} \rangle \equiv \forall s, x. \Gamma(\Sigma, \rho, s, e) \cap A(\Sigma, \text{pre}) \subseteq C(\Sigma, \text{pst}). \tag{2}$$

It represents that the set of computations ω, which starts at the configuration (θ, s, e), with $s \in pre$ and a computation $\delta \in \omega$. If an execution terminates, then the final states belong to *pst*.

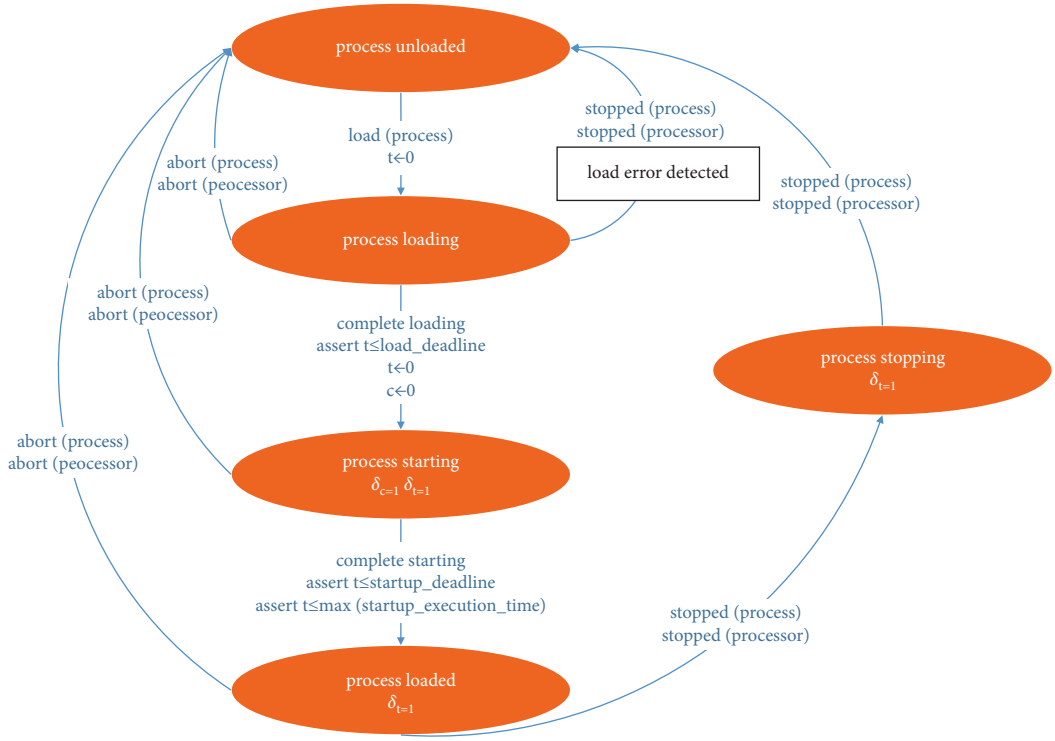

Figure 30: Process execution automaton.

sy-state ((*sy-st s*) *system*))=*SYS-OPERATIONAL*;
process∈ (*get-pros-bysys m system*);
(*pr-state* ((*pr-st s*) *process*))=*PRO-UNLOADED*;
(*pr-state* ((*pr-st t*) *process*))=*PRO-LOADING*;
gt = (*cur-time s*);
(*begin-time* ((*pr-st s*) *process*)) = (*cur-time s*);
(*action-begin-time* ((*pr-st s*) *process*)) = (*cur-time s*);
(*cur-time t*) = (*cur-time s*) + (*elapsed-time* ((*pr-st t*) *process*));
(*cur-time t*) = (*action-begin-time* ((*pr-st s*) *process*)) + (*elapsed-time* ((*pr-st t*) *process*));
(*exe-time* ((*pr-st t*) *process*)) ≤ (*elapsed-time* ((*pr-st t*) *process*))

Figure 31: The process execution Load semantic.

5.3.2. Invariant Verification. The core of the correctness proof shows the invariance of states between components generated from it. The proof proceeds by induction on the former, and actually, it is long and contains many technicalities. In many cases, we would like to show that the AADL execution model preserves certain data invariants. Since the Behavior annex may not be a closed system; that is, a state may be changed by its environment or conditions. So that the reachable states of Behavior annex depend on both the initial states and the environment. A Behavior annex with static information Σ is defined as follows:

$$\forall s_0, x_0\delta \cdot \delta \in \Gamma(\Sigma, \rho, s_0, e_0) \cap A(\Sigma, \text{inits}) \longrightarrow \left(\forall i < \text{length}(\delta) \cdot \text{invars}\left(s_{\delta_i}\right)\right). \tag{3}$$

The above formula demonstrates that it starts up from a set of initial states *init*, and it will preserve an invariant *inv* if its reachable states satisfy the predicate.

In this definition, δ denotes an arbitrary computation of ρ from a set of initial states *inits* and under an environment R. It requires that all states in δ satisfy the invariant *invars*.

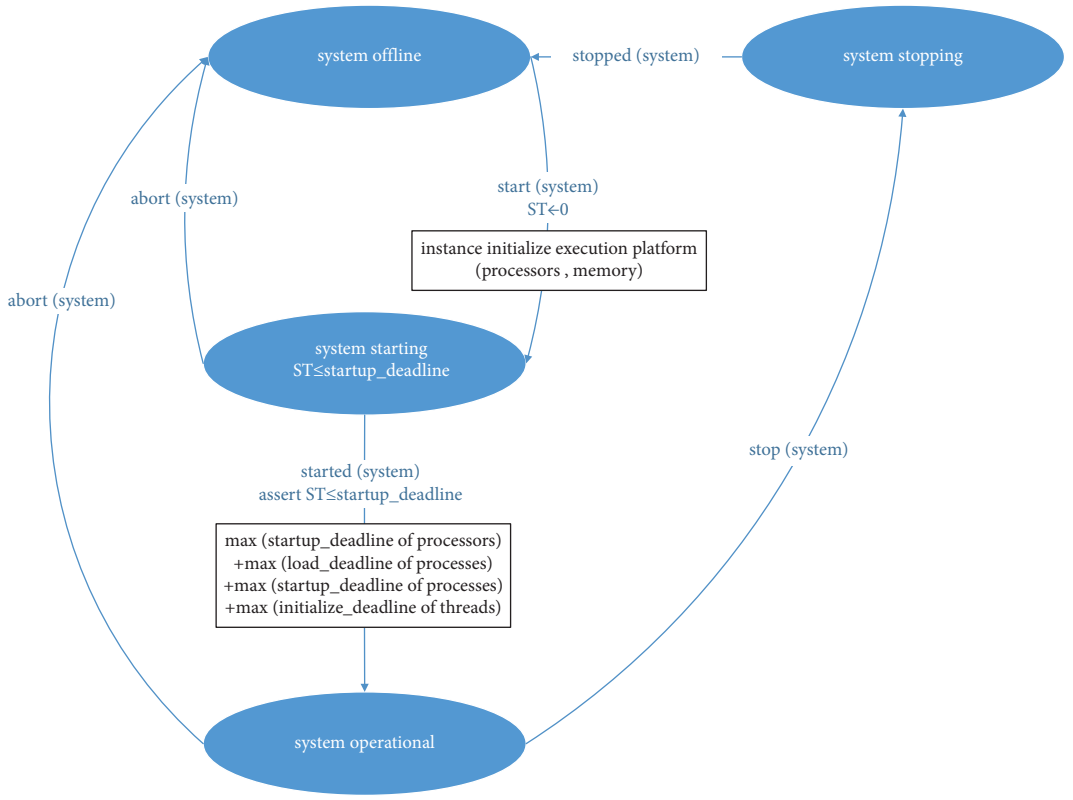

Figure 32: System execution automaton.

(*processor-state s*) = *PCOR-STARTED*;
(*sy-state* ((*sy-st s*) *system*))=*SYS-STARTING*;
(*sy-state* ((*sy-st t*) *system*))=*SYS-OPERATIONAL*;
process∈(*get-pros-bysys m system*);
gt = (*cur-time s*);
(*cur-time s*) = *0*;
(*begin-time* ((*sy-st s*) *system*)) = (*cur-time s*);
(*action-begin-time* ((*sy-st s*) *system*)) = (*cur-time s*);
(*cur-time t*) = (*cur-time s*) + (*elapsed-time* ((*sy-st t*) *system*));
(*cur-time t*) = (*begin-time* ((*sy-st s*) *system*)) + (*elapsed-time*
((*sy-st t*) *system*));
(*cur-time t*) = (*action-begin-time* ((*sy-st s*) *system*)) + (*elapsed-time*
((*sy-st t*) *system*));
(*exe-time* ((*sy-st t*) *system*)) ≤ (*elapsed-time* ((*sy-st t*) *system*));
(*elapsed-time* ((*sy-st t*) *system*)) ≤ (*get-pro-loaddeadline-max m*
(*get-pros-bysys m system*))
+ (*get-pro-startupdeadline-max m* (*get-pros-bysys m system*))
+ (*get-thd-initialzedeadline-max m* (*get-thds-bypro m process*));
(*elapsed-time* ((*sy-st t*) *system*)) ≤ (*the* (*sys-startup-deadline* (*the*
(*system-type.sys-properties* ((*system-tp m*) *system*)))))

Figure 33: The System execution Start_Complete semantic.

To show that *invars* is preserved by a system ρ, and it suffices to show the invariant verification theorem as follows. This theorem indicates that (1) the system satisfies its specification $\langle$inits, post$\rangle$, (2) *invars* initially holds in the set of initial states, and (3) each action transition as well as each environment transition preserves *invars*. Later, by the proof system, invariant verification is decomposed to the verification of individual executions.

Theorem 1. *(Invariant Verification). For formal specification ρ and Σ, a state set inits, and invars, if*

(i) $\Sigma \vdash \rho$ *sat* $\langle inits, post \rangle$.

(ii) $inits \subseteq \{s \cdot invars(s)\}$,

then invars is preserved by ρ, inits.

6. Case Study: Formalization of an ARINC653-Based System

Our work aims at the formal specification and verification in a system development based on the AADL, so we apply the proof system for the specification, the validation, and the verification of an ARINC653-based System. In Figure 34, we provide an example, which is based on the ARINC653 OS platform using AADL with its Behavior Annex specification. This example is adapted from the ARINC653 annex document for the AADLv2 and shows a system with two partitions. It shows the components involved in the modeling of the ARINC653 system and illustrates the mapping of ARINC653 concepts to the AADL.

In fact, the architecture is described as the client thread "a_client" for calls and communication of action: either do not need to wait on the calculation of long distance calls and finished to send, or due to server for HSER subroutine call and waiting for the results to the values, send the results and return to continue to wait for the next execution among them, the thread a data port connection between tasks, each thread internal use behavior to define its specific behavior, describe the action to perform with state systems conditions and order.

6.1. Formal Transformation of AADL into Isabelle/HOL. We define several modeling rules of model transformation from the AADL model into Isabelle/HOL specification for the next step about formal validation and verification. The model transformation rules of AADL are specified with a set of corresponding rules between AADL and Isabelle/HOL in a way to obtain a modular specification, and a part of the transformation rules is described as follows.

6.1.1. Transformation of Components and Connections. Transform components and connections to datatypes in Isabelle/HOL.

6.1.2. Transformation of Properties and Features. Transform properties and features of components to the predefined records type as definitions in Isabelle/HOL. Notice that, if there is any subcomponent as a existed component in the other component, it is considered an abbreviation instead of secondary definition.

6.1.3. Transformation of Behavior Annex. Transform Behavior annex specification comprises some sophisticated procedures, and transformation rules are as follows:

(i) Transform variables in a Behavior annex to the predefined records type as definitions in Isabelle/HOL.

(ii) Transform states in a Behavior annex corresponding to initial, complete, and final states to denote the current state.

(iii) Transform transitions in Behavior annex as the predefined records type, and transform guards and actions in a transition to conditions and actions list as definitions. Assemble the elements representing transitions to one compositional definition, which comprise all the state transitions of a behavior specification.

As depicted in Figure 35, we show the segmental transformation code for the example thread in Figure 34.

6.2. Formal Instantiation, Validation, and Verification in Isabelle/HOL. This section introduces the next formal steps of the example model after transformation into Isabelle/HOL as well as the validation and verification with its proof system. In this section, we use the instantiation of the AADL example to formally specify and verify the properties of the system model.

6.2.1. Instantiation. The basic transformation rules have been listed in Section 6.1, and we can use it to abstract an example of AADL model (see Figure 34) in Isabelle/HOL. In the implementation of AADL in Isabelle/HOL, we use record to create the framework, where components of AADL are represented as parameters and assumptions of record. Records are the Isabelle/HOL's approach for dealing with parametric datatype. Every component of the same type inside the system model can be mapped and encapsulated into an instantiation by Isabelle/HOL specification, and the component type and implementation are instantiated. In the last stage of modeling, we can integrate datatype to type variable as parameter and get the concrete AADL model code in Isabelle/HOL. For instance, the instantiation of the process type is implemented by the mapping function as follows:

primrec *process-type-map*: *ExProcess* (ExPort; ExDataaccess;

'Subpaccess; 'Parameter) process-type

where *pro-tp1: process-type-map partition1-process* = *partition1-process-type* |

pro-tp2: process-type-map partition2-process = partition2-process-type

6.2.2. Validation. The part of rules for grammar have been listed in Section 4.2, and we rewrite the validation rules as 47 definitions in Isabelle/HOL. After the formal description of rules, we reach the validation phase to check the grammar of the AADL model above. In our work, we use the validation rules code to check whether the model from the

```
thread a_server
features
  long :  provides subprogram access long_computation
    { Behavior_Properties :: Subprogram_Call_Protocol => LSER; };
  short :  provides subprogram access send_result
    { Behavior_Properties :: Subprogram_Call_Protocol => HSER; };
properties
  Dispatch_Protocol => timed;
  Period => 100 ms ;
end a_server;
thread implementation a_server . i
subcomponents
  local_result : data result_type . i;
 connections
  cnx1: data access  local_result -> local_result.result;
  cnx2: data access  local_result -> short.result;
annex  behavior_specification {**
  states
    s0 :  initial complete final state;
    s1 :  complete state;
    s2 :  complete state;
  transitions
    s0 -[  on dispatch  long ] -> s1;
    s1 -[  on dispatch  ] -> s2  timeout 60ms;
    s1 -[  on dispatch timeout  ] -> s2 { local_result.status : = 0 };
    s2 -[  on dispatch  short ] -> s20 { send_result! (local_result, local_result) };
**};
end  a_server . i;
```

Figure 34: AADL example of the ARINC653-based system: a typical thread with Behavior annex.

transformation satisfies a given property specified with temporal logic. Since the lemmas of validations are consistent with the integrating model, the proof obligations for the validation rule are proven immediately after unfolding the definitions of the precondition, postcondition, and relations. After applying the conditional and the grammar rules on the components, only the proof of the verification of each lemma body is left. Using these auxiliary lemmas, the postcondition is proven immediately by applying the properties over multisets. All the lemmas of validations are similarly proven, we omit the details here and the interested reader can refer to the Isabelle/HOL sources. We present an example of validation 7th as follows.

6.2.3. Safety Verification. Safety represents "nothing bad will happen," which comprises reachability or properties expressed in the form of finite state automata by invariance. After transforming the AADL abstract model to a target concrete model, we use the proof system (see Section 5.3) to verify its trace refinement and reachability properties.

Trace refinement checks "whether the abstract behavior trace of an implementation satisfies its abstract behavior trace of a specification." For instance, an assertion for trace refinement compares the whole abstract behaviors of a given action with another action, that is, whether there is a succeed relationship. For one of actions, the refinement analysis of actions is executed as follows.

Reachability refers to the ability to get from one state to another with one or multiple events. For instance, definition action-reach state shows the concrete state's reachability of the action for Behavior annex:

Only with thread inputs and outputs without interior actions, the above definitions are used to verify whether all abstract behaviors refine the outside abstract behaviors, and whether the system reaches the goal of state.

```
datatype ExThread = a-client | a-server
datatype ExBehaviorAnnex = ba-a-client | ba-a-server
definition long-a-server-conf :: (ExData, ExSubprogram, Ex-
Thread, ExProcess, ExSystem) subpaccess-conf
  where long-a-server-conf ≡ (|spac-name = "long",
                               spac-dir=PROVIDES,
                               spac-right =None,
                               spac-queueprotocol =None,
                               spac-queuesize =None,
                               spac-queue =None,
                               spac-obj =Some (SCSubp long-computation)|)
abbreviation local-result ≡ result-type
definition ba-a-server-conf :: ('s, 'ExDispatcher, 'Port, ExDataac-
cess, ExSubpaccess, ExParameter, ExData, ExSubprogram, Ex-
Thread, ExProcess, ExSystem) behavior-annex-conf
        where ba-a-server-conf ≡ (|ba-states={s0-ba-a-server,
s1-ba-a-server, s2-ba-a-server},
                                  ba-trans={tran1-ba-a-server,tran2-ba-a-server,
tran3-ba-a-server,tran4-ba-a-server},
                               ba-vars={},
                               ba-name="Dispatcherehavior-specification"|)
definitiona-server-impl :: (ExConnection, 'Subprogramcalls, ExDa-
ta, ExSubprogram, ExThread, ExProcess, ExSystem, ExBehaviorAn-
nex) thread-impl
  where a-server-impl ≡ (|impl-subcomps= {SCData local-result},
                         impl-conns = {cnx1-a-server, cnx2-a-server},
                         impl-name = "a-server.i",
                         thd-spcalls={},
                         thread-impl.thd-properties=None,
                         thread-impl.thd-ba=Some ba-a-server|)
```

Figure 35: The segmental code of Thread in Isabelle/HOL.

7. Evaluation and Conclusion

Our work presents a method of the description of AADL and a methodology of model transformation from a comprehensive subset of AADL to Isabelle/HOL. To specify this transformation, a preliminary analysis and comprehension of AADL and Isar/Isabelle/HOL languages are necessary and reveal the need to take into account the various parts of the language: structural, execution model, and its semantics description. Then, we use Isabelle/HOL as the specification, instantiation, validation, and verification system to conduct proofs against the properties of grammar and semantic in the structured proof language Isar, allowing for proof text naturally understandable for both humans and computers.

7.1. Evaluation Results. All derivations of our proofs have passed through the Isabelle/HOL proof kernel. The total development of our framework has ≈ 1280 lines of Isabelle/HOL specification and proof (LOSP). The concrete syntax of AADL consists of ≈ 630 LOSP, and the semantic of AADL consists of ≈ 650 LOSP. The two parts of specification and proof are completely reused in AADL . We use ≈ 750 LOSP to develop our validation system and ≈ 500 LOSP to develop the verification system including the formalization of 47 grammar rules. Finally, we develop ≈ 3300 LOSP for three case studies of AADL system model, which is ARINC653-based. We find two grammatical mistakes in the second case study and summarize that the instantiation in Isabelle/HOL has ≈ 3 times as much code as the lines of the AADL model.

7.2. Conclusion and Future Works. Different from the majority of AADL formal approaches above, our proposition aim at defining a formal executable semantics of a comprehensive AADL subset to allow the instantiation, validation, and verification of behavioral and temporal properties. Besides, the considered AADL subset consists of both software and hardware AADL components with a significant set of temporal and queuing AADL properties. The considered subset covers fundamental features that can be used in more realistic applications rather than "without behavior" and "model transformation into other languages"

approaches. Our experience is encouraging, but much more works remain ahead. First, increasingly larger AADL subsets should be considered to face complex applications such as shared variables by several threads with subprogram access, complex scheduling, etc. in the future works. Second, we need more complex industrial applications to examine our theory and the toolset, adjust our schema, and revise the technical architecture and implementation details, so as to realize our object that increase the confidence of safety-critical software. We plan to extend the AADL in Isabelle/HOL to support more structures and stepwise refinement. Third, we need to verify more properties like the rules of model transformation conforms to semantics equivalence, the satisfaction of the noninfluence, etc. And the following important perspective concerns are compilation aspect from AADL to C language as our next step.

Data Availability

Data sharing is not applicable to this article as no datasets were generated or analyzed during the current study.

Conflicts of Interest

The authors declare no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Acknowledgments

The part of previous work on formal specification has been published at IOP Science as a conference paper [26]; the whole work of this article is based on and improved on it. This research was supported (in part) by the formal verification project for the microkernel operating system.

References

[1] RTCA and EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification," *RTCA DO-178*, 2011.

[2] SAE, "Architecture Analysis & Design Language (AADL)," *AS 5506C-2017, Version 2.2*, 2017.

[3] SAE, "Architecture Analysis & Design Language (AADL) Annex," *AS 5506/2-2011*, vol. 2, 2011.

[4] I. Malavolta, L. Patricia, M. Henry, P. Patrizio, and T. Antony, "What industry needs from architectural languages: a survey," *IEEE Transactions on Software Engineering*, vol. 39, pp. 869–891, 2013.

[5] T. Bourke, L. Brun, P. E. Dagand, X. Leroy, M. Pouzet, and L. Rieg, "A formally verified compiler for Lustre," *Acm Sigplan Notices*, vol. 52, pp. 586–601, 2017.

[6] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, 2009.

[7] R. Krebbers, X. Leroy, and F. Wiedijk, *Formal C semantics: CompCert and the C standard. Interactive Theorem Proving*, Springer International Publishing, Berlin, Germany, 2014.

[8] N. Tobias, M. Wenzel, and L. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer-Verlag, Berlin, Germany, 2002.

[9] T. Nipkow and G. Klein, *Concrete Semantics*, Springer Publishing Company, New york, NY, USA, 2014.

[10] L. C. Paulson, *ML for the Working Programmer*, Cambridge University Press, Cambridge, UK, 2nd edition, 1996.

[11] M. Wenzel, "The Isabelle/Isar reference manual.," 2021, https://isabelle.in.tum.de/doc/isar-ref.pdf.

[12] A. E. Rugina, K. Kanoun, and M. Kaaniche, "The ADAPT tool: from AADL architectural models to stochastic petri nets through model transformation," in *Proceedings of the European Dependable Computing Conference (EDCC)*, Kaunas, Lithuania, May 2008.

[13] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated verification of AADL-specifications using UPPAAL," in *Proceedings of the 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, Omaha, NE, USA, October 2012.

[14] R. Jean-Francois, B. Jean-Paul, F. Mamoun, C. David, and T. Dave, "Modes in asynchronous systems," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems IEEE*, Belfast, UK, March 2008.

[15] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of AADL models via a translation into synchronous programs," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, pp. 134–143, ACM, Salzburg Austria, September 2007.

[16] B. Bernard, J. P. Bodeveix, C. Chaudet, and S. D. Zilio, "Formal verification of AADL specifications in the topcased environment," in *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe '09)*, pp. 207–221, Springer-Verlag, Brest, France, June 2009.

[17] C. Yang, D. Yunwei, Z. Fan, A. Ehsan, and G. Bin, "Formal semantics of AADL models with machine-readable CSP," in *Proceedings of the IEEE/ACIS 11th International Conference on Computer and Information Science IEEE*, Shanghai, China, June 2012.

[18] Z. Feng, Z. Yongwang, M. Dianfu, and N. Wensheng, "Formal verification of behavioral AADL models by stateful timed CSP," *IEEE Access*, vol. 5, 2017.

[19] M. Y. Chkouri, B. Marius, S. Joseph, and R. Anne, *Translating AADL into BIP - Application to the Verification of Real-Time Systems*, Models in Software Engineering Springer-Verlag, Toulouse, France, 2009.

[20] L. Besnard, B. Adnan, G. Thierry et al., "Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony," *Science of Computer Programming*, vol. 106, pp. 54–77, 2015.

[21] Z. Yang, "From AADL to timed abstract state machine: a certified model transformation," *Journal of Systems & Software*, vol. 93, no. 2, pp. 42–68, 2014.

[22] K. Hu, T. Zhang, Z. Yang, and W. T. Tsai, "Exploring AADL verification tool through model transformation," *Journal of Systems Architecture*, vol. 61, no. 3-4, pp. 141–156, 2015.

[23] H. Mkaouar, Z. Bechir, J. Mohamed, and H. Jérôme, "An ocarina extension for AADL formal semantics generation," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pp. 1402–1409, New York, NY, USA, April 2018.

[24] T. Abdoul, "AADL execution semantics transformation for formal verification," in *Proceedings of the Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on IEEE*, Washington, DC, USA, March 2008.

[25] P. C. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral AADL models in real-time Maude," in *Formal Techniques for Distributed Systems. FMOODS 2010, FORTE 2010. Lecture Notes in Computer*

Science, J. Hatcliff and E. Zucca, Eds., vol. 6117, Berlin, Germany, Springer, 2010.

[26] Yu Tan, D. Ma, and L. Qiao, "Towards formal specification for AADL with behavior annex in Isabelle," *IOP Conference Series: Earth and Environmental Science*, vol. 769, no. 4, Article ID 42016, 2021.