Towards Formal Specification for AADL with Behavior Annex in Isabelle

Yu Tan¹,*, Dianfu Ma¹, Lei Qiao²
¹School of Computer Science and Engineering, Beihang University, Beijing 100191, China
²Beijing Institute of Control Engineering, Beijing 100190, China
*Corresponding author: tanyu@act.buaa.edu.cn

Abstract. The analysis of safety-critical systems designed by architectural languages such as AADL (Architecture Analysis and Design Language) is a challenging research topic. In such a context, formal methods become an advocated practice in software engineering for rigorous analysis. Moreover, they are applied on specific formalisms to be analyzed on dedicated tools. This paper studies the comprehensive formal specification for AADL language, in particular supporting major components of AADL and Behavior Annex. The presentation of this specification and modeling is the aim of this paper. This work is illustrated with a ARINC653 case study. As a study case, this work develops an AADL model from an ARINC653, specify a set of critical properties of the model and perform formal modeling in Isabelle/HOL.

Keywords: formalization, specification, AADL, Isabelle

1. Introduction
In safety-critical domains such as avionics, aerospace, automotive, and defense, 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 DO178C [1] which stipulates analysis, testing and certification activities. Formal methods have become a recommended practice in the safety-critical domain.

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 such systems as an assembly of software components mapped onto an execution platform. 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 industry [3]. AADL has been studied in several projects for different modeling, analysis, simulation, compilation, extension and formal verification. AADL language adopts formal modeling concepts for the description of software and execution platform architecture, so it is used to design and analyze the software and hardware architecture of performance critical real-time systems. As a supplement of run-time environment in terms of distinct components and their interactions, the standard AADL Behavior annex [4] 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 are available at the modeling phase, it is especially effective for model-riven design of complex embedded real-time systems.

The AADL language provides an abundant 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 of model analysis and verification. Unfortunately, AADL is a textual and graphical language, which means it is a semi-formal modeling language. It lacks formal specification, and this severely limits both unambiguous communication among model developers, and the development of simulators and formal analysis tools, so it cannot be directly used for formal verification. In this work, we choose Isabelle/Isar/HOL [5], a tool suit (within its functional language) that gathers specification of AADL language and models, and also the verification towards AADL model for the future work. Isabelle/Isar language provides a readable grammar and a convenient way to produce the proofs.

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

- Different from transforming AADL into other formal model languages, our work takes an approach by formally specifying AADL languages.
- Our work focus on safety-critical software, so the considered AADL subset consists of both software and hardware AADL components with complex state transitions is comprehensive and that can be used in more realistic applications.
- We perform formal specification of the AADL model and perform formal instantiation of the realistic AADL model.

In this context, we aim at the comprehensive formal specification of AADL core language (software part) with its behavior annex. The remainder of this paper is organized as follows: SECTION II summarizes the related work of formalisms of AADL, SECTION IV, we describe the concept of AADL along with its Behavior annex and Isabelle/HOL, 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 V, we present the syntax of our selection in Isabelle/HOL. SECTION VI, we present a case study. SECTION VII gives the conclusions and future directions.

2. RELATED WORK
AADL lacks formal specification, 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. The one method is often based on model transformation into different languages without (or barely with) behavior annex such as Petri nets [6], Lustre [7], CSP [8]. These approaches are contented with the AADL described in its own standard which is enough to formally simulate the system and verify a set of behavioral properties. The others represents the work about the model transformation of AADL language with its Behavior annex such as BIP [9], Signal [10], TASM [11]. They define a transformation into the BIP language, and then the almost only specializes in behavior and analysis by using and mapping AADL behavioral models, such as IF [12], Real-time Maude [13]. These AADL formal approaches mainly consider different AADL subsets(with or without annexes) and carry on formal verification with existing tools like UPPAAL, Tina, Polychrony, etc. They often define a model transformation to implement whole AADL model certification instead of AADL language itself. Our work considers several resource information in the transformation, and the theorem prover is used to prove the methodology, i.e. the correctness of the translation.
3. AADL AND ISABELLE/HOL NOTATIONS

3.1. AADL
AADL is a textual and graphical language based on a component-centric model, it is used to model, specify and analyze architectures (included software and hardware part) of safety-critical and real-time embedded systems. AADL defines the system architecture as a set of interconnected components that hierarchically describes a system as a hierarchy of software and hardware components and offers a set of predefined component categories as follows:

- Software components: Data, Subprogram, Subprogram group, Thread, Thread group, Process, and their Types, Implementations, Features, Connections, Properties.
- Execution platform components (Hardware components): Processor, Memory, Bus and Device.
- System composites: which represent composite sets of software and execution platform components.
- Annex Subclauses: which allow annotations expressed in a sublanguage to be attached to 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.

3.2. Isabelle/HOL
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 which abbreviates Higher-Order Logic [14]. Isabelle is implemented in ML [15]. This has influenced some of Isabelle/HOL’s concrete syntax Isabelle/Isar [16], 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 formalisation and the output, moreover, also will invoke the theorem prover’s code generator and ran 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 restrict itself to the core of Isabelle (simply typed Lambda calculus with ML-style polymorphism and inductive datatypes). 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.

4. OVERVIEW

4.1. Selection of Core AADL
In this paper, our work focuses on the specification and analysis of the software components of systems, the most of execution platform components in AADL (virtual processors, memory, buses, virtual buses and devices etc.) is 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 AADL code, a 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. Which, a processor component is an abstraction of hardware and software that is responsible for scheduling and executing threads that are bound to it, and a software system represents an assembly of interacting application software.

Features are consisted with port, access and parameter features in our work. In addition, features can be combined with properties and our work can support some temporal and queuing properties, such like Dispatch_Protocol (periodic, sporadic, timed), Period, Queue_Processing_Protocol(FIFO,
LIFO), Queue Size, Elapsed Time, Execution Time and Scheduling Protocol. Our work supports connections between port, parameter, data access and subprogram access. Our work focuses on the indispensable properties which depend on the specific components.

4.2. 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 component to make design and analysis easier. A Behavior annex instance is defined on the vocabulary consisting of its private variables behavior_variable, of 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, 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 statement a behavior specification subclauses only can be added in a thread, and the behavior specification subclauses describes the thread which the behavior specification subclause belongs to, since the execution of whole system at one processor is actually the execution of one thread.

5. ABSTRACT SYNTAX AND SPECIFICATION

Our work provides the abstract syntax of the considered AADL in 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 paper, to keep the paper 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 main syntax are classified in the following subsection.

5.1. Features and Connections

Features are a part of components 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, some of them are defined as the following TABLE 1 and 2.

| Table 1. The syntax of Feature and Connection |
|-----------------------------------------------|
| feature ::= port | data_access | subprogram_access | parameter |
| connection ::= connection_identifier |
|                ( port_connection | access_connection |
|                          parameter_connection ) |
|                [ '{' { property_association }+ '}' ] |
|                [ in_modes_and_transitions ] |
Table 2. The code of Feature and Connection in Isabelle

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

5.2. 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, 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 is presented as TABLE 3.

Table 3. The syntax of Component type and Implementation base

| Syntax                               |
|--------------------------------------|
| **type_base** ::= [ type_name ] [ features { feature }+ ] |
| **impl_base** ::= impl_name [ connections { connection }+ ] |
| [ subcomponents { subcomponent }+ ] |

By default we consider one component Type h as 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 as TABLE 4.

Table 4. The code of Component type and Implementation in Isabelle

| Code                                                |
|-----------------------------------------------------|
| **record** `('Port, 'Dataaccess, 'Subpaccess, 'Parameter) type-base =` |
| type-features :: `('Port, 'Dataaccess, 'Subpaccess, 'Parameter)` |
| type-name :: string |
| **record** `('Connection, 'Data, 'Subprogram, 'Thread, 'Process, 'System) impl-base =` |
| impl-subcomps :: `('Data, 'Subprogram, 'Thread, 'Process, 'System)` |
| impl-conns :: 'Connection set |
| impl-name :: string |
5.3. **Software Subcomponents**

Software subcomponents represent components contained within another software component and contained in a component implementation may be instantiations of component implementations that contain subcomponents themselves. As the statement of the core AADL selection in SECTION IV, the software subcomponents are specified as Data, Subprogram, Thread, Process and System. Their instances are presented as TABLE 5.

**Table 5.** The syntax of Subcomponents.

| Component Type          | Syntax Description                           |
|-------------------------|---------------------------------------------|
| Data Type               | `data_type ::= type_base [data_properties]` |
| Data Implementation     | `data_impl ::= impl_base [data_properties]` |
| Subprogram Type         | `subprogram_type ::= type_base [subprogram_properties]` |
| Subprogram Implementation| `subprogram_impl ::= impl_base [subprogram_properties]` |
| Subprogram Call         |                                           |
| Thread Type             | `thread_type ::= type_base [thread_properties] [behavior_annex]` |
| Thread Implementation   | `thread_impl ::= impl_base [thread_properties] [subprogramcalls] [behavior_annex]` |
| Process Type            | `process_type ::= type_base [process_properties]` |
| Process Implementation  | `process_impl ::= impl_base [process_properties]` |
| System Type             | `system_type ::= type_base [system_properties]` |
| System Implementation   | `system_impl ::= impl_base [system_properties]` |

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. Due to space constraints, we show only the code snippets for the Data subcomponent as the following TABLE 6.

**Table 6.** The partial Code of Subcomponents in Isabelle

```isabelle
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

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

5.4. **Behavior Annex**

As the SECTION IV, a behavior specification subclauses is a part of a thread, and it describes the thread which the behavior annex belongs to. The Behavior annex is composed of variable set, state set, transition set and its private information (like its name, ports of its parent component, etc.), and its elements are united by its transitions.
The transitions can be described the behavior as a state transition system linked with guards by some conditions and actions. A behavior transition consists of its name, source state, destination state, guard condition and actions. The actions can be classified as basic actions and action blocks in the transitions. The action blocks are in the form of sequences or sets. Every single action block is like imperative language and can be defined as conditionals and loops. The guards combined of conditions are in the transitions are explicitly classified as dispatch conditions and execution conditions. The Behavior annex instance is presented as TABLE 7.

Table 7. The syntax of Behavior annex.

| behavior_annex ::= [variables { behavior_variable }+] [states { behavior_state }+] [transitions { behavior_transition }+] |
| behavior_transition ::= [ trans_identifier [ [ trans_priority ] ] : ] source_state_identifier { , source_state_identifier }+ -- [ guard_condition ]→ destination_state_identifier [ action_block ], |
| guard_condition ::= execute_condition | dispatch_condition |
| execute_condition ::= logical_expression | no_others |
| dispatch_condition ::= on dispatch [ trigger_condition ] [ frozen ( frozen_ports ) ] |
| 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 }+ |

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 totally 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 as the following TABLE 8.
Table 8. The syntax of Behavior annex.

| Datatype | Description |
|-------------------------------|-------------|
| 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 |
| 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 |
| 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 | DispatchCond-TriggerLogicExp |
| DispatchCond-Stop | DispatchCond-Stop 'Dispatcher × Event |
| DispatchCond-Timeout | 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 |

Notice: Dispatcher is a predefined type, 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 as a function computing outputs from the value of its input ports. The time consumption of an action is directly modelled as a Time attribute.
5.5. Whole Comprehensive 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.

Table 9. The partial Code of Whole Model in Isabelle

| 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 } |

6. CASE STUDY: SPECIFICATION OF AN ARINC 653 BASED SYSTEM
In this paper, our work aims at the formal specification in a system development based on the AADL language, so we apply the proof system for the specification of an ARINC 653 based System. We specify three system models based on ARINC653. For reasons of space, we provide an example which is based on ARINC653 OS platform using AADL with its Behavior Annex specification in FIGURE 1. This example is adapted from the ARINC653 annex document for the AADL v2 and shows the use of a LSER subprogram call between a client thread and a server thread. The client thread does not need to wait for the completion of the call to a long remote calculation. Result value is later returned thanks to a HSER subprogram call to the server. It shows the components involved in the modeling of ARINC653 system and illustrates the mapping of ARINC653 concepts to the AADL.

6.1. Formal Transformation of AADL Model into Isabelle/HOL
We define several modeling rules of the transformation from AADL model into the specification in Isabelle/HOL for the next step about formal validation and verification in the future work. The model transformation rules of AADL is 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. Transformation of components and connections. Transform components and connections to datatypes in Isabelle/HOL. 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 as an abbreviation instead of secondary definition. Transformation of behavior annex specification comprises some sophisticated procedures as the following transformation rules:

- Transform variables in a behavior annex to the predefined records type as definitions in Isabelle/HOL.
- Transform states in a behavior annex corresponding to initial, complete, and final states to denote the current state.
- 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 definitions which comprise all the state transitions of a behavior specification.

As depicted in FIGURE 1, we show the segmental transformation code for the example thread in TABLE 10.

**Table 10.** The segmental code of Thread in Isabelle

```isabelle
datatype ExThread = a-client | a-server

datatype ExBehaviorAnnex = ba-a-client | ba-a-server

definition long-a-server-conf :: (ExData, ExSubprogram, ExThread, ExProcess, ExSystem) subpaccess-conf
  where long-a-server-conf ≡ (| spac-name=''long'', spac-dir=PROVIDES, spac-right=None, spac-queueprotocol=None, spac-queue=None, spac-obj|=Some (SCSubp long-computation) |)

abbreviation local-result ≡ result-type

definition ba-a-server-conf :: ('s, ExDispatcher, 'Port, ExDataaccess, ExSubpaccess, ExParameter, ExData, ExSubprogram, ExThread, ExProcess, ExSystem) behavior-annex-conf
  where ba-a-server-conf ≡ (| ba-states=fs'-ba-a-server, s1-ba-a-server, s2-ba-a-server, ba-trans=ftran1-ba-a-server, tran2-ba-a-server, tran3-ba-a-server, tran4-ba-a-server, ba-vars=fg, ba-name=''Dispatchehavior-specification'' |)

definition a-server-impl :: (ExConnection, 'Subprogramcalls, ExData, ExSubprogram, ExThread, ExProcess, ExSystem, ExBehaviorAnnex) thread-impl
  where a-server-impl ≡ (| impl-subcomps= fSCData local-resultg, impl-conns= fcnx1-a-server, cnx2-a-server, impl-name=``a-server:i'' , thd-specalls=fg, thread-impl:thd-properties=None, thread-impl:thd-ba=Some ba-a-server |)
```

6.2. Instantiation

The basic transformation rules have been listed above, we can use it to abstract an example of AADL model 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. Record are the Isabelle/HOL’s approach for dealing with parametric datatype. Every component of same type inside the system model can be mapped and encapsulated into a instantiation by Isabelle/HOL specification, and the component type and implementation are instantiated respectively. In 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 process type is implemented by the mapping function as follows:

| Table 11. The partial Code of mapping function in Isabelle |
|----------------------------------------------------------|
| primrec thread_type_map :: "ExThread⇒ (Port, ExDataaccess, ExSubpaccess, ExParameter, ExData, ExSubprogram, ExThread, ExProcess, ExSystem, ExBehaviorAnnex) thread_type" |
| where thd_tp1: "thread_type_map a_client = a_client_type" |
| thd_tp2: "thread_type_map a_server = a_server_type" |
| primrec thread_impl_map :: "ExThread⇒ (ExConnection, 'Subprogramcalls, ExData, ExSubprogram, ExThread, ExProcess, ExSystem, ExBehaviorAnnex) thread_impl option" |
| where thd_im1: "thread_impl_map a_client = None" |
| thd_im2: "thread_impl_map a_server = Some a_server_impl" |
| primrec thread_ba_map :: "ExThread⇒ ExBehaviorAnnex option" |
| where thd_ba1: "thread_ba_map a_client = Some ba_a_client" |
| thd_ba2: "thread_ba_map a_server = Some ba_a_server" |
| primrec behavior_annex_map :: "ExBehaviorAnnex ⇒ ('s, ExDispatcher, 'Port, ExDataaccess, ExSubpaccess, ExParameter, ExData, ExSubprogram, ExThread, ExProcess, ExSystem) behavior_annex_conf" |
| where thd_ba1: "behavior_annex_map ba_a_client = ba_a_client_conf" |
| thd_ba2: "behavior_annex_map ba_a_server = ba_a_server_conf" |
| 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 |

7. EVALUATION AND CONCLUSION

Our work presents a method of 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 description. Then we use Isabelle/HOL as the specification and instantiation system to program against the properties of grammar in the structured proof language Isar, allowing for text naturally understandable for both humans and computers.

Evaluation results. All derivations of our proofs have passed through the Isabelle/HOL proof kernel. The total development of our framework has ≈ 39300 lines of Isabelle/HOL specification and proof (LOSP). The abstract syntax of AADL language consists of ≈ 630 LOSP. The two parts of specification and proof are completely reused in AADL language respectively. We use ≈ 3300 LOSP for three case studies of AADL system model based on ARINC653. We find two grammatical
mistakes in the second case study, and summarize that the instantiation in Isabelle/HOL has \( \approx 3 \) times as much code as the lines of the AADL model.

**Conclusion.** Different from the majority of AADL formal approaches above, our proposition aim at defining a formal specification of a comprehensive AADL subset to allow the instantiation 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.

**Future works.** Our experience is encouraging, but much more works remain ahead. First, increasingly larger AADL subsets should be considered to face complex applications in the future works, such as shared variables by several threads with subprogram access, complex scheduling, etc. Second, we need more complex industrial applications to examine our method and toolset, so as to realize our object that increase the confidence of safety-critical software. Third, the following important perspective concerns will be, which is about the semantics of AADL, as the next step in our work.

**Acknowledgements**

This research is supported (in part) by the National Key Research and Development Program of China under Grant No. 2018YFB0203802.

**References**

[1] Adams, C. . "DO-178C nears finish line, with credit for modern tools and technologies." Military & Aerospace Electronics 21.11(2010): p.22,24-28.

[2] AS5506C: SAE Architecture Analysis and Design Language (AADL) Version 2.2. 2017.

[3] Malavolta, I., et al. "What Industry Needs from Architectural Languages: A Survey." IEEE Transactions on Software Engineering 39.6 (2013): 869-891.

[4] AS5506/2: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2. 2011.

[5] Nipkow, T., L. C. Paulson , and M. Wenzel. "Isabelle/HOL: A Proof Assistant for Higher-order Logic." Springer-Verlag (2002).

[6] Rugina, A. E., K. Kanoun , and M. Kaaniche . "The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation." Seventh European Dependable Computing Conference IEEE Computer Society, 2008.

[7] Jahier, E., et al. "Virtual Execution of AADL Models via a Translation into Synchronous Programs." (2007): 134.

[8] Yang, Chunxin., et al. "Formal Semantics of AADL Models with MachineReadable CSP." 2012 IEEE/ACIS 11th International Conference on Computer and Information Science IEEE, 2012.

[9] Chkouri, Mohamed Yassin, et al. "Translating AADL into BIP - Application to the Verification of Real-Time Systems." Models in Software Engineering Springer-Verlag, 2009.

[10] Besnard, Loiec , et al. "Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony." Science of Computer Programming 106. aug.1(2015):54-77.

[11] Yang, Zhibin. "From AADL to timed abstract state machine: a certified model transformation." Journal of Systems & Software 93.2 (2014): 42-68

[12] Abdoul, Thomas , et al. "AADL Execution Semantics Transformation for Formal Verification." Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on IEEE, 2008.

[13] PC Ölveczky, A. Boronat, and J. Meseguer. "Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude."Springer-Verlag Springer-Verlag, 2010.

[14] Nipkow, Tobias, and G. Klein. "Concrete Semantics." (2014).

[15] Paulson, L. C. . "ML for the Working Programmer." Cambridge Univ. Press (1996).
[16] Ballarin, C. . "Locales and locale expressions in Isabelle/Isar." Types for Proofs & Programs Lncs 3085(2003): 34-50.