IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 2, 2020, accepted April 7, 2020, date of publication April 14, 2020, date of current version April 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2987972

Formal Verification of AADL Models by Event-B

ABEER SAEED ABDO HADAD[1], CHUNYAN MA[1], AND ADEEB ABDULWAKEEL OBADI AHMED[2]
[1]School of Software, Northwestern Polytechnical University, Xi'an 710072, China
[2]Department of Mechanical Engineering, Xidian University, Xi'an 710072, China

Corresponding author: Chunyan Ma (machunyan@nwpu.edu.cn)

This work was supported by the Key Laboratory Project of Aviation Science Foundation under Grant 20175553028 and Grant 20185853038.

ABSTRACT AADL is widely used to depict the architecture and behavior of real-time safety-critical systems such as avionics and aerospace. The development of these systems has strict requirements for building fault-free systems. Formal verification is frequently applied to verify the critical properties of the systems, such as safety and liveness; however, the formal verification is not supported by AADL. Model transformation is commonly applied to provide formal semantics; hence, the AADL model can be verified by a language that supports formal verification in addition to the ability to cover all AADL model behavior. Event-B, with its proof obligation, is increasingly used to model and verify safety-critical systems. This paper presents the transformation of the AADL model into Event-B, which captures most AADL components and behavioral actions to be effective in the verification of current real-time systems models. Then, we define theorems and invariants of safety and liveness properties to be proven by using the RODIN platform. To demonstrate the efficacy of our method, we model the AADL of movement authority (MA) control of the Chinese Train Control System, transform the AADL model to Event-B and verify its crucial properties.

INDEX TERMS Model transformation, AADL, behavior annex, Event-B, proof obligation, invariant, theorem proving.

I. INTRODUCTION

The development of safety-critical systems such as real-time systems has rapidly increased in various domains such as health, transport and automotive. From this perspective, building a fault-free system has become an essential need through their development life cycle. One of the most commonly used approaches in the design phase is the formal verification, which can ensure critical properties early in the development phases.

The Architecture Analysis and Design Language (AADL) [1] is a description language that represents the architecture of the system as a hierarchy of decomposed interacting software and hardware components. The AADL describes the constructional aspect of models in addition to non-functional requirements such as timing. Although the AADL provides efficient support for modeling the safety-critical real-time systems, it must be formalized to make the model convenient for formal verification.

Event-B [2] is a formal language that describes concurrent safety-critical systems. The Event-B model structure and behavior are basically described by first-order logic and set theory, so they have rigorous mathematical descriptions. Event-B provides a primary characteristic called refinement which enables the gradual development of the system. Therefore The AADL elements can gradually be transformed to make possible the traceability of Event-B against AADL and make the transformed model extensible. In addition, Event-B has a key feature called proof obligation, which provides mathematical proof of the properties according to a set of rules. RODIN platform tool [3] introduces support for Event-B modeling, automatic creation and proving the rules.

Several studies have already been adopted to transform the AADL model to a formal language such as CSP [4]–[6], Fiacre [7], [8], BIP [9], Maude [10], [11], LNT [12], [13] and TASM [14], [15]. However, all of these works consider only a small AADL subset. Therefore, they cannot cover most AADL components and their behavioral actions, which are increasingly included in real-time system models.

In this paper, we present a method to transform the AADL model into Event-B. We use the UML class diagram to serve as an intermediary between AADL and Event-B; in order to make AADL more clear, appropriate for Event-B, and to ease the traceability of the AADL element's details. Since the

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

AADL is used to model the real-time systems as a hierarchy, those systems' models contain different relationships among the elements, which also has many details such as features and properties. The Event-B which is based on the set theory, is used to model the discrete systems as states transition. Therefore, first, we present a description of AADL by the UML class diagram, then this AADL class diagram is transformed into Event-B. A large AADL subset is included to cover multiple safety and liveness properties and make the proposed method more effective in the verification of current real-time systems models. We define a set of clear mapping rules that involve all details of the transformation of AADL to Event-B. The contributions of our paper are as follows:

1) We consider a large AADL subset, including threads (periodic, aperiodic, timed, sporadic), remote subprogram call and access connection.
2) We present the correctness of the transformation, ensure preservation of the semantics and determine whether the transformation is deterministic or terminating.
3) We verify both safety and liveness properties that have been verified by the existing studies. Our method also proposes the verification of AADL constraint preservation and shared data deadlock.

The remainder of this paper is organized as follows. Section II-A briefly presents some basic concepts related to the AADL. Section II-B introduces Event-B and explains the features of Event-B. Section III-A defines the AADL subsets. Section III-B presents our methodology and the mapping rules. Section IV introduces the verification of the transformation correctness. Section V presents the case study. Section VI presents the related work, Section VII provides the discussion, and Section VIII concludes.

II. BACKGROUND

A. AADL

The Architecture Analysis and Design Language introduces a well-known description of the hardware and software components and their interaction with each other to propose complete systems. The AADL is separated into three categories: software, hardware, and composite. Each category consists of multiple components; these components with type and implementation classifiers describe the architecture of the system. The composite category involves the system, which enables the integration of all components into one unit. The hardware category consists of the processor, device, bus, and memory.

The software category consists of the process, thread, thread group, data, and subprogram. Each software component declaration is divided into two classifiers: type and implementation. The type includes features and properties. The features determine how the component communicates with other system components. The features include port, provides subprogram access, requires subprogram access and requires data access. The component implementation declaration may include subcomponent, mode and behavior annex.

The behavior annex provides a sublanguage extension to link the behavior specifications to the AADL components. The behavior annex aims to depict the internal behavior of component implementations such as subprogram calls and synchronization protocols for client-server architectures as a state transition system with guards and actions.

B. EVENT-B

Event-B [2], [16] is used to formally model safety-related systems in terms of state transitions. Event-B consists of two main constructs: *context* and *machine*. *Context* represents the static part of the system model, whereas the *machine* represents the behavioral part. The *machine* is related to the *context* by the *see* relationship, so the machine can access the context contents. Multiple machines can *see* one context, and a machine can *see* multiple contexts. The *context* contains the sets s, constants c, axioms A and theorems THM. The *set* usually describes the system attributes that can be defined as a group of elements. The *constant* defines the elements of the *set* or the system variables that do not change through system behavior. Sets and constants are constrained by *axioms*. *Theorems* define the properties derivable from *axioms*. The *machine* involves state variables V, invariants I and set of events evi. The state variables are constrained by the *invariant*. The event describes the state transition and has the following two forms:

$$ev \doteq any\ x\ where\ G\ then\ Act. \tag{1}$$

$$ev \doteq when\ G\ then\ Act. \tag{2}$$

The event's form (1) represents the form of an event when parameters (x) is defined, whereas the form (2) represents the form of an event when parameters (x) is not defined. Generally, x is the event parameters, G is the guard and Act is the action. The guard represents the essential condition of the event to be enabled and perform the action Act.

Event-B presents two mechanisms to reduce the modeling complexity: machine refinement and context extension. Refinement helps the designer start modeling within abstract specifications and then gradually add the model details. Moreover, the invariants that are proven at the abstract level are maintained through refinement.

In addition to the refinement and extension, Event-B introduces decomposition [17], [18] to address the modeling complexity and provides the modeling of parallel and concurrent systems [19], [20]. Decomposition is a mechanism that splits the model into smaller sub-models. Two styles of decomposition are proposed: shared-variables and shared-events. In this paper, the shared-variables style is considered where variables are divided into internal and external. The sub-model shares the external variables, whereas the internal variables are private for each one. Events are also internal and external, where external events are used for sub-model communication. Both internal variables and events are refined as usual in Event-B, whereas external variables are difficult to refine.

TABLE 1. Basic Event-B mathematics notations in this paper.

Notation	Description
$\in$	Set membership
$\mathbb{N}$	The set of natural numbers (nonnegative integers)
BOOL	BOOL = {TRUE, FALSE}
:=	"becomes equal to" $x := e$ means assign to the variable x the value of the expression e
$\rightarrow$	Denotes a total function. If $f \in X \rightarrow Y$ and $x \in X$, then $f(x)$ is defined.
$\emptyset$	empty set
$\mapsto$	$a \mapsto b$ (a maps to b) is the ordered pair of a and b
$\Rightarrow$	logical implication: $P \Rightarrow Q$, if P is true then Q is true, but $Q \Rightarrow P$ is not necessarily true
$\forall$	Universal quantification. For all x, P is true
$\exists$	Existential quantification. There exists x such that P is true.
$\mathbb{P}$	$P(S)$ which is the set of all subsets of S.
$\triangleright$	Range restriction: $r \triangleright s$ is the subset of r in which the range is restricted to the set s.

Proof obligation (PO) [21] represents the backbone of Event-B to demonstrate the correctness of the model regarding some behavioral semantics. POs verify various model properties in terms of the invariant or theorems. Multiple kinds of POs are automatically generated by RODIN Platform. In the following, we only present the related POs kinds within our scope of research.

1) Invariant Preservation PO (INV): in the abstract machine, for each invariant and event, POs are defined to prove that each invariant is preserved by each event.
2) Refinement PO
   If machine M refines machine N, machine M is called the concrete machine, and machine N is the abstract machine; the following POs are defined,
   a) Invariant Preservation PO (INV): verify that each concrete event preserves both concrete and abstract invariants.
   b) Simulation PO (SIM): each action in the abstract event is simulated by the corresponding concrete actions, which ensures that what the concrete events execute does not contradict what the corresponding abstract event executes.
3) Context and Machine Theorem PO (THM): the PO of each theorem is automatically created to ensure that a stated context and machine theorems are provable from the axioms.

The choice of Event-B is explained by several reasons as follows: **(i) the ability to describe all architectural and behavioral semantics of AADL** by means of available mechanisms such as refinement and decompositions. **(ii) By using the Event-B invariants**, the AADL constraints are transformed, and the preservations of these properties are verified by each reachable state. The invariants also provide the ability to include more critical properties to verify. **(iii) Enable the gradual development of the system** by refinement. Since the refinement enables the designer to gradually transform AADL elements to present the traceability of the AADL against Event-B, the designer can also gradually transform new properties and components that have been added to the AADL model without re-transforming the entire model. In addition, Event-B mathematically provides verification of the consistency among refinements levels, i.e., the proven properties in the abstract level are preserved through refinement. **(iv) RODIN platform has many integrated useful tools** for modeling, such as ProB [22], which allows animation and model checking, so that the transformed Event-B model can easily be validated. **(iv) Event-B has no ambiguous grammatical structure**; therefore, the one-to-one transformation and verification of the transformation correctness are easily applied.

III. OUR METHOD

Our working methodology is divided into three phases: In the first phase, a subset of AADL is selected, and the strategy of behavioral semantics is determined using the UML class diagram. In the second phase, a set of mapping rules is presented to transform the AADL class diagram model to Event-B. In the third phase, strong invariants and theorems are defined to describe the properties to be verified. The corresponding proof obligations are automatically generated and proven by the RODIN platform.

A. AADL SUBSET

In this paper, we only consider the following AADL software components: process, thread, subprogram and data components. These components can sufficiently describe the basics of the system behavior. To more clearly describe the AADL element relationships, we define the elements using the UML class Diagram, which is the best model to define different elements, their details and the relationship among them. The AADL UML class diagram is illustrated in Figure 1, where class refers to the AADL elements, method refers to the element actions, attributes refer to the features, properties and element details, and class relationships refer to the AADL element relationships. The classes in the AADL system are as follows:

1) Port class
   Port is a logical data and control connection point among AADL threads and processes. The *Port* class in Figure 1 represents the AADL port, which is inherited by *IN_port* and *OUT_port* classes. There are three types of input/output ports: data, event and event data ports, which are represented by *IN_DT/ OUT_DT, IN_DEV/ OUT_DEV*, and *IN_EV/ OUT_EV* classes respectively.
   The *IN_DT, OUT_DT, OUT_EV*, and *OUT_DEV* classes have two similar attributes: *(port_variable, port_state)*, where,
   - *port_variable* refers to the port variable. The *port_variable* has different data types: DATA in the *IN_DT* and *OUT_DT*, EVENT in the *OUT_EV* and EVENT_DATA in the *OUT_DEV*.

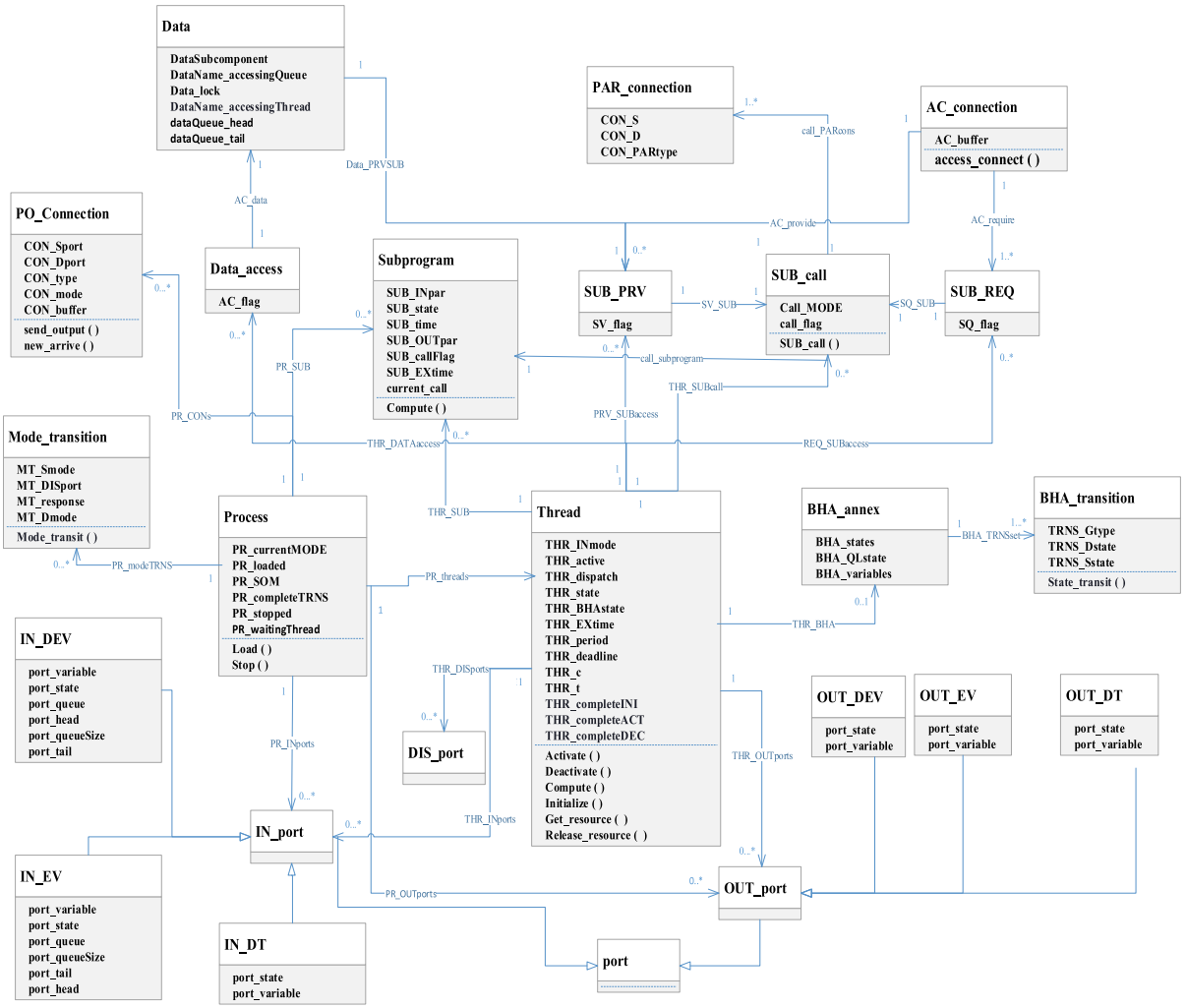

FIGURE 1. UML class diagram of AADL subset.

- *Port_state* is a Boolean variable that indicates whether the state of the port variable is fresh.

The *IN_EV* and *IN_DEV* classes have six attributes *(port_queue, port_head, port_tail, port_variable, port_state, port_Qsize)*, where,

- *port_queue* refers to the port queue.
- *port_head* and *port_tail* are integer variables to apply the FIFO queue protocol.
- *port_variable* is a variable to receive the new incoming value if the queue is empty and the dequeued value if the queue is not empty.
- *port_state* is a Boolean variable that indicates the state of *port_variable* whether it is a new value or not.

2) PO_connection class

The port connection describes the linkage between the ultimate source and the destination component's port. The *PO_connection* class in Figure 1 represents the AADL port connections. The *PO_connection* has the attributes *(CON_Sport, CON_Dport, CON_mode, CON_type, CON_buffer)*, where

- *CON_Sport* is the source port.
- *CON_Dport* is the destination port.
- *CON_mode* $\in$ *modes* is the *in mode* clause.
- *CON_type* $\in$ *connection_type* is the connection type *(IMMEDIATE, DELAYED, NONE)*, *IMMEDIATE* and *DELAYED* are for the data port connections. The *NONE* is for the event and event data port connections.
- *CON_buffer* is the connection buffer.

The *PO_connection* class has two methods: *send_output* and *new_arrive*. The *send_output* method represents the *send_output* service call. The *new_arrive* method represents the arrival of the *EVENT, DATA* or *EVENT DATA* to the destination port.

3) Mode_transition class

Mode describes the runtime operational state. Different modes may be declared in the threads and processes.

Mode transition specifies the mode switching between two different modes.
Mode_transition class in Figure 1 represents the AADL process mode transition. The *Mode_transition* class has attributes *(MT_Smode, MT_Dmode, MT_DIS-port, MT_reponse)*, where,

- *MT_Smode* is the ultimate source mode.
- *MT_Dmode* is the destination mode.
- *MT_DISport* ∈ *IN_port* is the process port that triggers the mode switching.
- *MT_reponse* ∈ *MT_TRNStype* refers to the *Mode_Transition_Reponse* property, which determines whether the transition is *EMERGENCY* or *PLANNED*.

The *Mode_transition* class contains the *transit* method, which represents the mode transition actions.

4) Subprogram class
The subprogram is the executable unit called by the threads or other subprograms. It consists of input parameters and output parameters.
The *Subprogram* class in Figure 1 represents the AADL subprogram component. The *Subprogram* class has the attributes *(SUB_INpar, SUB_OUTpar, SUB_call, SUB_state, SUB_EXtime, SUB_time, SUB_CALLflag, current_call)*, where

- *SUB_INpar* is the set of subprogram input parameters.
- *SUB_OUTpar* is the set of subprogram output parameters.
- *SUB_state* ∈ *State* is a variable that indicates the current state of the subprogram.
- *SUB_EXtime* is a constant that represents the maximum execution time of the subprogram.
- *SUB_time* is a time variable that calculates the subprogram execution time.
- *SUB_CALLflag* is a Boolean variable, which has the value of *TRUE* if the subprogram has been called; otherwise, it holds the value of *FALSE*.
- *current_call* ∈ *SUB_call* refers to the current executed call by the subprogram.

The *Subprogram* class contains the *compute* method, which represents the subprogram execution actions.

5) PAR_CON class
The parameter connection describes the data flow between the subprogram parameters and the caller component.
The *PAR_CON* class in Figure 1 represents the subprogram parameter connections. The *PAR_CON* class has attributes *(CON_S, CON_D, CON_PARtype)*, where,

- *CON_S* is the connection source.
- *CON_D* is the connection destination.
- *CON_PARtype* ∈ *PARTYPE* is the type of connection (input or output).

The *PAR_CON* class contains the *PAR_connect* method and represents the parameter connection actions.

6) SUB_call class
The thread or subprogram that must call the subprogram has a subprogram call sequence. The *SUB_call* class in Figure 1 represents the subprogram call; it has two attributes: *Call_flag* and *Call_mode*. *Call_flag* is a Boolean variable that indicates whether the call has been activated or not. *Call_mode* represents the *in mode* clause. The *SUB_call* class has two association relationships with *Subprogram* and *PAR_CON* classes. The association relationship with the *Subprogram* class is represented by the *Call_subprogram* attribute to describe the called subprogram. The *Call_PARcons* association relationship connects the *SUB_Call* with *PAR_CON* to refer to the set of parameter connection of this call.
The *call* method in *SUB_call* class represents the subprogram call actions.

7) SUB_REQ, SUB_PRV and AC_connection classes
In the case of a remote subprogram call, a thread can call a subprogram that is a subcomponent of another thread. The called thread (server) has the *provides subprogram access* feature, and the caller thread (client) has the *requires subprogram access* feature. These two features are connected via access connection. The *SUB_REQ, SUB_PRV* and *AC_connection* classes in Figure 1 represent the *requires subprogram access* feature, *provides subprogram access* feature and access connection, respectively. The *SUB_REQ* class has only the *SQ_flag* boolean attribute. This attribute holds the value of *TRUE* if a new call has been sent; otherwise, it remains *FALSE*. The *SUB_REQ* class is connected to the *SUB_call* class by an association relationship. This relationship is represented by *SQ_call* to indicate the sent subprogram call by the caller thread. The *SUB_PRV* class has only an *SV_flag* Boolean attribute. This attribute holds the value of *TRUE* if a new call has been received; otherwise, it remains *FALSE*. *SUB_PRV* is connected to *SUB_call* via the *SV_call* association relationship. This relationship represents the received subprogram call. The *AC_connection* has the *AC_buffer* attribute referring to the access connection buffer. *The AC_connection* has two association relationships with *SUB_REQ* and *SUB_PRV*, which are represented by *AC_provide* and *AC_require*, respectively. The *access_connect* method in the*AC_connection* represents the connection actions.

8) DataName and Data_access classes
Data subcomponents represent static data (class) in the AADL source text. By the *requires data access* feature declaration, the Data will be shared and accessed by different components. *DataName* class in Figure 1 represents the AADL data subcomponents. The *DataName* class contains the following attributes:

- *DataName_subcomponents* is the data subcomponent of the data component. Each data subcomponent is represented by a *DataName* class attribute.
- *DataName_lock* is a Boolean variable that indicates whether the data has been locked or not.
- *DataName_accessingQueue* is a queue that contains the list of the waiting components that require to access data during data locking.
- *DataName_accessingThread* is a variable that indicates the current thread that locks the data.
- *dataThread_head* and *dataThread_tail* are variables to apply FIFO dequeue protocol on *DataName_accessingQueue*.

The Data component may have a *requires subprogram access* feature. This feature is represented by the *Data_SUB* association relationship with the *Subprogram* class.

The *Data_access* class represents the *requires data access* feature. The *Data_access* has a Boolean textit AC_flag attribute that is assigned to TRUE if the component has required to access the data; otherwise, it remains as FALSE.

The association relationship *AC_data* connects the *Data_access* class to the *DataName class* to represent the accessed data.

9) BHA_annex and BHA_transition classes

Thread implementation may contain the behavior annex to provide the thread behavior's refinement. The *BHA_transition* class in Figure 1 represents the AADL behavior annex state transitions. The *BHA_transition* contains two attributes: *(TRNS_Sstate, TRNS_Dstate)*, where

- *TRNS_Sstate* is the ultimate source state.
- *TRNS_Dstate* is the destination state.

The *BHA_annex* class represents the AADL behavior annex. It has two attributes *(BHA_states, BHA_QLstate)*, where,

- *BHA_states* is the set of behavior annex states
- *BHA_QLstate* is the state qualification to the thread states

The *BHA_annex* class is connected to the *BHA_transition* via the *BHA_TRNSset* association relationship to represent the set of state transitions.

The *state_transit* method in the *BHA_transition* class indicates the state transition actions

10) Thread class

The thread represents a schedulable unit that can simultaneously execute with other threads. The thread component type may contain properties and features such as *port* and the *provides subprogram access*, the *requires subprogram access* and the *requires data access* features. Thread implementation may have subprogram call and behavior annex.

A thread goes through five states (INITIAL, AWAITING_DISPATCH, AWITING_MODE, SUSPEND, FINIAL) and changes its state through different actions. The *Thread* class in Figure 1 represents the AADL thread. The Thread has multiple attributes *(THR_active, THR_dispatch, THR_state, THR_BHAstate, THR_EXtime, THR_deadline, THR_period, THR_c, THR_t, THR_completeINI, THR_comp-leteACT, THR_completeDEC)*, where,

- *THR_active* is a Boolean variable that indicates whether the thread is active in the current mode or not.
- *THR_dispatch* is a Boolean variable that indicates whether the thread has been dispatched or not.
- *THR_state* $\in$ *Thread_state* is a variable that refers to the current thread state.
- *THR_BHAstate* is a variable that refers to the current thread behavior annex state.
- *THR_EXtime* is a constant that refers to the value of maximum thread *computation_execution_time* property.
- *THR_deadline* is a constant that refers to the thread deadline property.
- *THR_period* is a constant that refers to the thread period property.
- *THR_c* and *THR_t*. *THR_t* is the time variable that is used in the timed, sporadic and periodic threads to calculate the time before thread dispatching. *THR_c* is a time variable to calculate the time during thread dispatching.
- *THR_INmode* indicates the modes where the thread is active.
- *THR_completeINI* is a Boolean variable that indicates whether the thread has completed initialization or not.
- *THR_completeACT* is a Boolean variable that indicates whether the thread has completed activation or not.
- *THR_completeDEC* is a Boolean variable that indicates whether the thread has completed deactivation or not.

The *Thread* class has nine different association relationships *(THR_INports, THR_OUTports, THR_DISports, THR_SUB, THR_DATAaccess, THR_SUBcalls, PRV_SUBaccess, REQ_SUBaccess, THR_BHA)*, where

- *THR_INports* connects the *Thread* to the *IN_port* class to represent the set of thread's in ports.
- *THR_OUTports* connects the *Thread* to the *OUT_port* class to represent the set of thread's out ports.
- *THR_DISports* connects the *Thread* to the *DIS_port* class to represent the set of thread's dispatch ports.

- *THR_SUB* connects the *Thread* to the *Subprogram* class to represent the set of subprogram subcomponent.
- *THR_DATAaccess* connects the *Thread* to the *Data_access* class to represent the set of the *requires data access* features.
- *REQ_SUBaccess* connects the *Thread* to the *REQ_SUB* class to represent the set of the *requires subprogram access* features.
- *PRV_SUBaccess* connects the *Thread* to the *PRV_SUB* class to represent the set of the *provides subprogram access* features.
- *THR_BHA* connects the *Thread* to the *BHA_annex* class to represent the behavior annex.

The *Thread* class contains the *Initialize, Activate, Deactivate, Compute, Finalize, Get_resourece*, and *Release_resource* methods, which indicate the Initialization, Activation, Deactivation, Computation, Finalization, Get_resource and Release_resource actions, respectively.

11) Process class

Process represents an enforced virtual address space at runtime. Process must contain at least one thread as a subcomponent. The AADL process has three main actions: process loading, process stopping and process aborting. The *Process* class in Figure 1 represents the AADL process. The Process class has multiple attributes *(PR_modes, PR_currentMODE, PR_loaded, PR_stopped, PR_SOM, PR_completeTRNS, PR_waiting-Thread)*, where

- *PR_modes* $\in$ *modes* is the set of process contained modes.
- *PR_currentMODE* is a variable that refers to the current mode.
- *PR_loaded* is a Boolean variable that indicates whether the process completes loading or not.
- *PR_stopped* is a Boolean variable that indicates whether the process completes stopping or not.
- *PR_SOM* is a Boolean variable that indicates whether the mode transition request has been received or not.
- *PR_completeTRNS* is a Boolean variable that indicates whether the mode transition has been completed or not.
- *PR_waitingThread* is a Boolean variable that indicates whether the process is waiting for the old mode threads to complete execution during mode switching or not.

Process class has five different association relationships *(PR_threads, PR_CONs, PR_modeTRNS, PR_INports, PR_OUTports)*, where

- *PR_threads* connects the *Process* to the *Thread* class to represent the set of thread subcomponent.

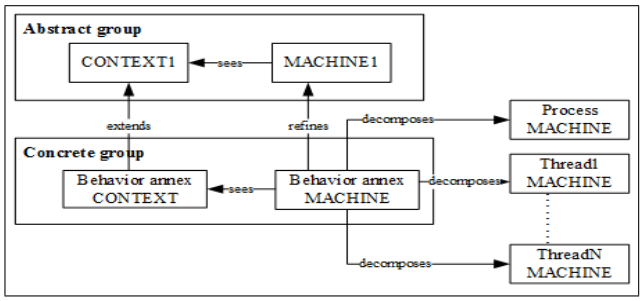

FIGURE 2. The general sketch map of the transformed Event-B model.

- *PR_CONs* connects the *Process* to the *PO_connection* to describe the connection in the process component.
- *PR_modeTRNS* connects the *Process* to the *Mode_transition* class to represent the set of mode transition that the process may have.
- *PR_INports* connects the *Process* to the *IN_port* class to represent the set of process in ports.
- *PR_OUTports* connects the *Process* to the *OUT_port* class to represent the set of process out ports.
- *PR_SUB* connects the *Process* to the *Subprogram* class to represent the set of subprogram subcomponent.

The *Process* class contains the *Load* and *Stop* methods, which indicate the process loading and stopping actions, respectively.

B. TRANSFORMATION OF THE AADL MODEL TO EVENT-B

In section III-A, we have represented the AADL components using the UML class diagram. This representation plays the role of an intermediary for the transformation into Event-B. We organize the AADL classes into two groups: concrete and abstract. The concrete group contains only the *BHA_annex* and *BHA_transition* classes, whereas the abstract group contains the remaining AADL classes. Figure 2 shows the general sketch map of our transformed Event-B model, where the abstract group *(CONTEXT1, MACHINE1)* contains the corresponding Event-B semantics of the abstract classes. The concrete group *(Behavior annex CONTEXT, Behavior annex MACHINE)* contains the corresponding Event-B semantics of the concrete classes. The two groups are connected via the *refines* and *extends* relationships. The concrete *Behavior annex MACHINE* is decomposed into several submodels according to the number of threads that the process may contain.

The *Process MACHINE* contains all variables and actions related to the *Process* class. Each *THREAD MACHINE* contains *Thread* attributes and methods. The communication among decomposed MACHINEs is established through shared variables. We start by the transformation of the abstract AADL group to the Event-B abstract group.

TABLE 2. Basics of mapping rules.

AADL class diagram	Event-B	Rule	Description
Class	Context constant	Rule 1	Map the AADL class to a context constant with axiom type of *className* $\in \mathbb{P}$*(CLASSNAME).*
Class inheritance relationship	Context axiom	Rule 2	Map the AADL class inheritance relationship to a context axiom of partition *(className, sub_class1, sub_class2,. . . . sub_classN).*
Class constant attribute	Context constant	Rule 3	Map the AADL class constant to a context constant with axiom type of *attributeName* $\in$ *className* $\rightarrow$ *attributeDATATYPE.*
Class association relationship	Context constant	Rule 4	Map the AADL class association relationship to a context constant with axiom type of *associationAttributeName*$\in$ *sourceClassName* $\rightarrow$ *DestinationClassName.*
Class variable attribute	Machine variable	Rule 5	Map the AADL class variables to a machine variable with invariant type of *attributeName* $\in$ *className* $\rightarrow$ *attributeDATATYPE.*
Class method	Machine event	Rule 6	Each AADL class method is mapped to Event-B machine events.

```
THR_EXtime
THR_deadline
THR_period
THR_INmode
AXIOMS
  axm01  :  THR_EXtime ∈ thread →Z
  axm02  :  THR_deaddline ∈ thread →Z
  axm03  :  THR_period ∈ thread →Z
  axm04  :  THR_INmode ∈ thread → P(Mode)
```

FIGURE 3. Corresponding Event-B of AADL Thread constant attributes.

```
THR_DISports
THR_SUB
THR_DATAaccess
THR_INports
THR_OUTports
PRV_SUBaccess
REQ_SUBaccess
AXIOMS
  axm05  :  THR_DISports ∈ Thread→P (DIS_port)
  axm06  :  THR_SUB ∈ Thread→P (Subprogram)
  axm07  :  THR_DATAaccess ∈ Thread→P (Data_access)
  axm08  :  THR_INports ∈ Thread→P (IN_ports)
  axm09  :  THR_OUTports ∈ Thread→P (OUT_ports)
  axm10  :  PRV_SUBaccess ∈ Thread→P(SUB_PRV)
  axm11  :  REQ_SUBaccess ∈  Thread→P(SUB_REQ)
```

FIGURE 4. Corresponding Event-B of AADL Thread association attributes.

Table 2 lists the basic mapping rules of the transformation approach. Then, the same approach is applied for the AADL concrete transformation.

a: TRANSFORMATION OF ABSTRACT CLASSES

1) **Transformation of AADL classes**
Rule 1 maps each AADL class to a context constant with the same name of the class. The constant is constrained by the axiom of *className* $\in$ $\mathbb{P}$*(CLASSNAME)*, where *CLASSNAME* is a set to define the className constant data type. For example, Thread $\in \mathbb{P}$(THREAD).
2) **Transformation of AADL class inheritance relationships.**
Rule 2 maps the inheritance relationship to the context axiom of *partition (className, sub_class1, sub_class2. . . sub_classN)*, where *sub_classi* is the class that inherits the *className* class.
3) **Transformation of the AADL class constant attributes.**
Rule 3 maps the class constant attributes to the context constant with the same name of attributes. This constant is constrained by the axiom of *attributeName* $\in$ *className* $\rightarrow$ *attributeDATATYPE*. If the attribute is an array, the axiom type becomes *attributeName* $\in$ *className* $\rightarrow$ $\mathbb{P}$*(attributeDATATYPE)*, where attributeDATATYPE represents the data type of the class attribute. For example, Figure 3 shows the transformation of the *Thread* class constant attributes.
4) **Transformation of AADL class association relationships**
Rule 5 maps the class association attributes to the context constant with the same name of the attributes. This constant is constrained by the axiom type of *associationAttributeName* $\in$ *sourceClassName* $\rightarrow$ *DestinationClassName*. If the association relationship is one-to-many, the axiom type becomes *associationAttributeName*$\in$ *sourceClassName* $\rightarrow$ $\mathbb{P}$ *(DestinationClassName)*. For example, Figure 4 shows the transformation of the *Thread* class association attributes.
5) **Definition of AADL classes' object.**
After the whole AADL classes, class constants, and attributes have been mapped, the class objects are defined. The class objects are mapped to the constant and corresponding axiom constraints. For example, Figure 5 shows the AADL examples and corresponding Event-B axioms, where *thread1, thread2, subprogram1, inport1, inport2, outport2, access1, access2, REQ1*, and *PRV1* have been defined as context constants.
6) **Transformation of AADL class variables attributes.**
Rule 4 maps the class variable attributes to machine variables. The variable has the name of *classobjectName_attributeName* to make each class object have its own variables. The variable are constrained by the invariant of *classObjectName_attributeName* $\in$ *className* $\rightarrow$ *attributeDATATYPE*. If the attribute is an array, the axiom type becomes *classObjectName_attributeName* $\in$ *className* $\rightarrow$ $\mathbb{P}$ *(attributeDATATYPE)*. For example, Figure 6 shows the

AADL thread example

```
data data1
end data1;
data implementation data1.i
end data1.i;
thread thread1
    features
        inport1: in data port;
        access1:  requires data access data1.i;
        ouport1: out event port;
        REQ1: requires subprogram access subprogram1.imp;
        properties
        compute_Execution_time => 2ms..3 ms;
        deadline => 4ms;
        Dispatch_Protocol=> periodic ;
        period=>1 ms;
 end thread1;
thread thread2
    features
        inport2: in data port;
        access2:  requires data access data1.i;
        PRV1: provides subprogram access subprogram1.imp;
        properties
        compute_Execution_time => 1ms..5 ms;
        deadline => 6ms;
        Dispatch_Protocol=> aperiodic ;
 end thread2;
 thread implementation thread2.im
   subcomponents
          sub1: subprogram subprogram1.imp;
 end thread2.im;
 subprogram subprogram1
 end subprogram1;
 subprogram implementation subprogram1.imp
 end subprogram1.imp;
```

The corresponding Event-B axioms

```
axm31   :   subprogram1 ∈ Subprogram
axm32   :   partition(Thread,{thread1},{thread2})
axm34   :   partition(Port,{inport1},{inport2},{outport3})
axm35   :   partition(IN_ports,{inport1},{inport2})
axm36   :   partition(OUT_ports,{outport3})
axm37   :   partition(IN_DT,{inport1})
axm38   :   partition(IN_EV,{inport2})
axm39   :   partition(OUT_EV,{outport1})
axm40   :   partition(Data_access,{access1},{access2})
axm41   :   REQ1 ∈ SUB_REQ
axm42   :   PRV1 ∈ SUB_PRV
axm43   :   THR_INports={thread1↦{inport1},thread2↦{inport2}}
axm44   :   THR_OUTports={thread1↦{outport3}, thread2↦∅}
axm50   :   THR_DISports={thread1↦∅,thread2↦{inport2}}
axm51   :   THR_SUB={thread2↦subprogram1}
axm52   :   PRV_SUBaccess={thread1↦REQ1}
axm53   :   REQ_SUBaccess={thread2↦PRV1}
axm54   :   THR_DATAaccess={thread1↦{access1},thread2↦{access2}
axm55   :   THR_EXtime={thread1↦3,thread2↦5}
axm56   :   THR_deadline={thread1↦4,thread2↦6}
axm57   :   THR_period={thread1↦1}
```

FIGURE 5. The Event-B semantic of AADL classes objects.

```
thread1_dispatch
thread1_state
thread1_active
thread1_t
thread1_c
INVARIANTS
  inv21   :   thread1_dispatch ∈ {thread1}→BOOL
  inv22   :   thread1_state ∈ {thread1}→Thread_state
  inv23   :   thread1_active ∈ {thread1}→BOOL
  inv24   :   thread1_t ∈ {thread1}→Z
  inv25   :   thread1_c ∈ {thread1}→Z
```

FIGURE 6. Corresponding Event-B of AADL classes' variables.

corresponding Event-B semantics of the *Thread* class variables attributes.

7) **Process class methods**
The Process *Load* and *Stop* methods are mapped to Event-B machine events. In this paper, we assume that the process is stopped and started loading when an event has been received. The Process *Load* method is mapped to two machine events: *processName_loading* and *processName_completeLoading*. The *processName_loading* receives the *LOAD_EVENT* and subsequently initializes all process ports. The *processName_completeLoading* updates the value of *PR_loaded* to *TRUE*. The *Stop* method is mapped to the *processName_stop* machine event. This event receives *STOP_EVENT* and marks the process as idle by updating the *PR_loaded* to *FALSE*.

8) **Thread class methods**
The five Thread class methods *(initialize, activate, deactivate, compute, and finalize)* are mapped to the machine events. The *initialize* method is mapped to two machine events: *threadName_initialization1* and *threadName_initialization2*. The *threadName_initializ-ation1* is enabled when the thread is a part of the initial mode and subsequently changes the thread state to *AWAITING_DISPATCH*. The *threadName_initialization2* changes the thread state to *AWAITING_MODE* if the thread does not belong to the initial mode. The activate and deactivate methods are mapped to the *threadName_activation* and *threadName_deactivation* events, respectively. In AADL, these two actions occur after the mode transition request has been received, which is represented by the *TRUE* value of *PR_SOM* variable in the transformed Event-B model. The *threadName_activation* event checks whether the thread is part of the current mode; then, the thread state is changed to *AWAITING_DISPATCH*. The *threadName_deactivation* is enabled when the thread is not part of the current mode; then, the thread state is changed to *AWAITING_MODE*.

Thread dispatch and timing

In the AADL models, timing constraints play an important role. However, Event-B only models the functional requirements, so no timing variables are declared.

In our proposed method, we assume that each timing variable is an integer variable, which is incremented by one when the system's clock increases by one, to represent the thread delay. The thread *compute* method is listed in Algorithm 1. This algorithm is described in Event-B by three machine events: *threadName_start_dispatch, threadName_execute,* and *threadName_complete_execution*. Figure 7 illustrates the Event-B representation of the thread timeline. The *threadName_start_dispatch* event's guard is specified according to the *dispatch_protocol* property, as described below.

- *Periodic* thread: $threadName\_t = THR\_period$
- *Aperiodic* thread: checks whether a new value has been arrived to the thread *THR_DISports* ports or not.
- *Timed* thread: $threadName\_t = THR\_period \vee$ checks the arrival of a new event to any *THR_DISports* port.

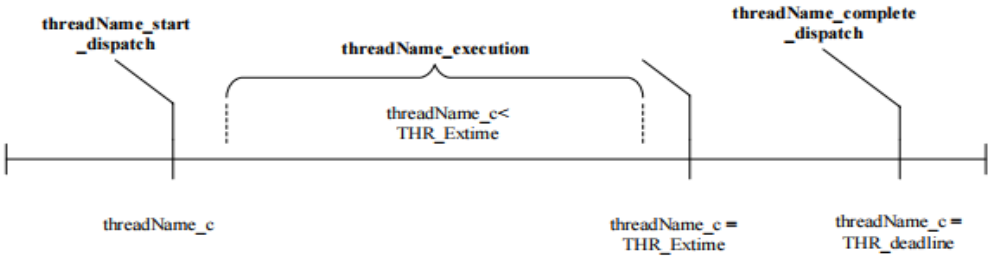

FIGURE 7. Thread computation timeline.

Algorithm 1 Thread *Compute* Description

loop
  if thread dispatch condition is *TRUE* **then**
    update *threadName_state* to *EXECUTION*
    update *thread_dispatch* to *TRUE*
  end if
  if *threadName_state* is *EXECUTION* **then**
    while *threadName_c* is less than *THR_EXtime* **do**
      increment *threadName_c* by one
    end while
    if *threadName_c* equal the *THR_EXtime* **then**
      send the value to out port
      if *CON_type* is *IMMEDIATE* **then**
        assign output port to *CON_buffer*
      end if
      update *threadName_state* to *AWAITING_DISPATCH*
      update *threadName_dispatch* to *FALSE*
      reset *threadName_c* to zero
    end if
  end if
  if the *PO_type* is *DELAYED* **then**
    assign output port to *CON_buffer*
  end if
  if thread is *aperiodic, Timed* or *sporadic* **then**
    if the dispatch port queue is not empty **then**
      dequeue an element and assign it to *portName_variable*
      update the *portName_state* to *TRUE*
    end if
  end if
end loop

AADL periodic thread example

```
thread EXP
    features
        in1: in data port;
    properties
        compute_Execution_time => 2ms..3 ms;
        deadline => 4ms;
        Dispatch_Protocol=> periodic ;
        period=>1 ms;
end EXP;
```

Corresponding Event-B events

```
EXP_clock   ≙
STATUS
 ordinary
WHEN
 grd1   :   EXP_state(EXP)=AWAITING_DISPATCH
 grd2   :   EXP_t(EXP)<THR_period(EXP)
THEN
 act1   :   EXP_t(EXP)≔EXP_t(EXP)+1
END

EXP_dispatch   ≙
STATUS
 ordinary
WHEN
 grd1   :   EXP_t(EXP)=THR_period(EXP)
 grd2   :   EXP_state(EXP)=AWAITING_DISPATCH
 grd3   :   EXP_dispatch(EXP)=FALSE
 grd4   :   EXP_c(EXP)=0
THEN
 act1   :   EXP_dispatch(EXP)≔TRUE
 act2   :   EXP_c(EXP)≔EXP_c(EXP)+1
 act3   :   EXP_state(EXP)≔EXECUTION
END

EXP_execution   ≙
STATUS
 ordinary
WHEN
 grd1   :   EXP_dispatch(EXP)=TRUE
 grd2   :   EXP_c(EXP)<THR_EXtime(EXP)
 grd3   :   EXP_state(EXP)=EXECUTION
THEN
 act1   :   EXP_c(EXP)≔EXP_c(EXP)+1
END

EXP_completeDispatch   ≙
STATUS
 ordinary
WHEN
 grd1   :   EXP_c(EXP)=THR_EXtime(EXP)
 grd2   :   EXP_dispatch(EXP)=TRUE
 grd3   :   EXP_state(EXP)=EXECUTION
THEN
 act1   :   EXP_state(EXP)≔AWAITING_DISPATCH
 act2   :   EXP_dispatch(EXP)≔FALSE
 act3   :   EXP_c(EXP)≔0
 act4   :   EXP_t(EXP)≔0
END
```

FIGURE 8. Periodic thread.

- *Sporadic*: *threadName_t* $\leq$ *THR_period* $\wedge$ checks the arrival of a new event to any *THR_DISports* port.

Then, the *threadName_start_dispatch* event updates *threadName_state* to *EXECUTION* and *threadName_dispatch* to *TRUE*. The *threadName_execute* event increments the *threadName_c* by one until it reaches the value of *thread_EXtime*. Consequently, *threadName_complete_execution* occurs. In the *aperiodic* thread, the *threadName_dequeue* event checks the queues of dispatch ports. If any queue is not empty, the thread continues executing.

For each *periodic, timed* and *sporadic* thread, the *threadName_clock* event is created to calculate the time. The *threadName_clock* event is enabled as long as the thread is active and waiting for dispatch. This event increments *threadName_t* by one until it reaches the value of *THR_period*; consequently, the event is disabled, and the thread becomes ready to be dispatched. Figure 8 shows an example of the *periodic* thread transformation.

Thread locking data access actions

The *Get_resource* and *Release_resource* methods' description is listed in Algorithm 2. The algorithm is represented in Event-B by four machine events: *threadName_getResource, threadName_releaseResource, recourseName_busy,* and *dataNameAccessQueue_dequeue*.

Algorithm 2 *Get_rsource* and *Release_resource* Methods Description

```
loop
  if threadName_dispatch is TRUE then
    if threadName_c equal one then
      if dataName_lock is FALSE then
        lock the data by updating the value of
        dataName_lock to TRUE
      else if dataName_lock is TRUE then
        add thread to waiting queue
        update threadName_state to SUSPEND
      end if
      if dataName_lock is FALSE and threadName_state
      is SUSPEND then
        lock the data by updating the value of
        dataName_lock to TRUE
        update threadName_state to EXECUTION
      end if
    end if
  end if
  if threadName_c equal THR_EXtime-1 then
    ulock the data by updating the value of
    dataName_lock to FALSE
  end if
end loop
```

The *threadName_getResource* event is enabled after the thread has been dispatched. This event checks the state of the data resource. If the data resource has not been locked, then the *dataName_lock* is updated to *TRUE*. The *recourseName_busy* event changes the thread state to *SUSPEND* if the data resource is locked by another thread and subsequently adds the thread to the *dataName_accessQueue*. The *threadName_releaseResource* event is enabled after the thread has completed its dispatch. This event updates *dataName_lock* to *FALSE*. Once the data resource is unlocked, the *dataNameAccessQueue_dequeue* event is enabled to allow other threads to lock the data according to the order of elements in *dataName_accessQueue*. Figure 9 shows the thread data access example and the corresponding Event-B events.

9) **Mode_transition class methods**

The *mode_transit* method description is listed in Algorithm 3. The algorithm is represented in Event-B by four machine events: *TRNSname_mode_transition1, TRNSname_mode_transition2, TRNSname_mode_tran-sition3*, and *TRNSname_mode_transition4*. The *TRNSname_mode_transition1* event is enabled when the value of *MT_response* is *EMERGENCY*.

This event checks the mode switch trigger port whether a new value has arrived or not. Then, the value of the *PR_SOM* variable is updated to *TRUE*. If the *MT_response* value is *PLANNED*, the *TRNSname_mode_transition2* event is enabled when old mode threads are still executing. The process waits for the threads in the old mode to complete execution. The *TRNSname_mode_transition3* event occurs after the old mode threads have completed their execution. Then, the value of the *PR_SOM* variable is updated to *TRUE*. The *TRNSname_mode_transition4* event checks whether the value of *threadName_completeACT* and *threadName_completeDEC* are *TRUE*. Then, the current

AADL thread data access example

```
data data1
end data1;
data implementation data1.i
end data1.i;

thread thread2
    features
        inport2: in data port;
        access2:  requires data access data1.i;
 end thread2;

 thread implementation thread2.im
 end thread2.im;
```

The corresponding Event-B events

```
thread1_getResorce
STATUS
 ordinary
WHEN
 grd1  :  thread1_dispatch(thread1)=TRUE
 grd2  :  thread1_state(thread1)=EXECUTION
 grd3  :  thread1_c(thread1)=1
 grd4  :  data1_lock(data1)=FALSE
THEN
 act1  :  data1_lock(data1)≔TRUE
 act2  :  data1_accessingThread(data1)≔thread1
END

data1_busy
STATUS
 ordinary
WHEN
 grd1  :  thread1_dispatch(thread1)=TRUE
 grd2  :  thread1_state(thread1)=EXECUTION
 grd3  :  thread1_c(thread1)=1
 grd4  :  data1_lock(data1)=TRUE
THEN
 act1  :  data1_accessQueue(data1)≔data1_accessQueue(data1)
          ∪ {data1Queue_tail(data1)↦thread1}
 act2  :  data1Queue_tail(data1)≔data1Queue_tail(data1)+1
 act3  :  thread1_state(thread1)≔SUSPEND
END

thread1_releaseResorce
STATUS
 ordinary
WHEN
 grd1  :  thread1_dispatch(thread1)=TRUE
 grd2  :  thread1_state(thread1)=EXECUTION
 grd3  :  thread1_c(thread1)=THR_EXtime(thread1)-1
 grd4  :  data1_lock(data1)=TRUE
THEN
 act1  :  data1_lock(data1)≔FALSE
END

data1Queue_dequeue
STATUS
 ordinary
WHEN
 grd1  :  data1_lock(data1)=FALSE
 grd2  :  data1_accessQueue(data1)≠∅
THEN
 act1  :  data1_lock(data1)≔TRUE
 act2  :  data1_accessQueue(data1)≔data1_accessQueue(data1)\
          {data1Queue_head(data1)↦data1_accessingThread(data1)}
 act2  :  data1_accessingThread(data1)≔data1_accessQueue(data1Queue_head(data1))
 act4  :  data1Queue_head(data1)≔data1Queue_head(data1)+1
END
```

FIGURE 9. Thread data access.

Algorithm 3 Mode_transition *transit* Method Description

```
loop
  if MT_reposnes is EMMREGENCY then
    update PR_SOM to TRUE
  end if
  if MT_reposnes is PLANNED then
    if current mode threads are still executing then
      wait threads to complete execution
    end if
    if threads complete the execution then
      update PR_SOM to TRUE
    end if
  end if
  if PR_SOM is TRUE then
    activate new mode threads
    deactivate old mode threads
  end if
  if threads complete activation and deactivation then
    update Process_currentMODE to the new mode
  end if
end loop
```

mode is changed to the new mode. Figure 10 illustrates the transformation of the mode transition.

10) **Subprogram, SUB_call, PAR_CON, and AC_connection classes' methods**

The *AC_connect* method is described in Algorithm 4. The algorithm is represented in Event-B by three Event-B machine events: *requiresFeatureName_call, providesfeatureName_call*, and *accessName_connection*. The *requiresFeatureName_call* event activates the subprogram call by updating the value of *SQ_flag* to *TRUE*. Then, the *accessName_connection* event receives the call, assigns it to *AC_buffer*, and marks the value as fresh by updating the value of the buffer *SQ_flag* to TRUE. Once the access buffer has a new value, the *providesfeatureName_call* event is enabled. Then, the server thread starts dispatching and subsequently calls the subprogram to be executed.

Figure 11 shows the transformation of the remote subprogram call.

The SUB_call *call* method, which represents the local subprogram call, is mapped to two machine events: *subprogramName_sendCall* and *subprogramName_call*. subprogramName_sendCall is enabled once the thread has been dispatched. Then, it sends the call to the subprogram by updating the value of *call_flag* to *TRUE*. *subprogramName_call* checks whether the value of *call_flag* is *TRUE*. Then, it updates *SUB_callFlag* to *TRUE* to start the subprogram execution. T he subprogram *compute* method is mapped to three Event-B machine events: *subprogramName_start, subprogramName_execute and subprogramName_complete*. The *subprogramName_start* is enabled once *SUB_callFlag* is updated to *TRUE*. Th *subprogramName_execute* and *subprogramName_complete* represent the subprogram computation delay.

AADL process mode example

```
process process1
    features
        in1: in event port;
        out1: out event port;
        modes
            mode1: initial mode;
            mode2: mode;
            mode1-[in1]->mode2 {Mode_Transition_Response=>PLANNED;};
            mode2-[in1]->mode1;
end process1;
    process implementation process1.imp
    subcomponents
        TH1: thread THR1.imp in modes(mode1);
        TH2: thread THR2.imp in modes(mode1);
    end process1.imp;
```

model to mode2 transition corresponding Event-B events

```
mode_transition1
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1  :  prm1∈dom(MD_Smode>{PR_currentMode(process1)})
 grd2  :  MT_Response(prm1)=EMERGNCY
 grd3  :  in1_state(in1)=TRUE
THEN
 act1  :  PR_SOM(process1)≔TRUE
END

mode_transition2
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1  :  prm1∈dom(MT_Smode>{PR_currentMode(process1)})
 grd2  :  MT_Response(prm1)=PLANNED
 grd3  :  in1_state(in1)=TRUE
 grd4  :  THR1_dispatch(THR1)=TRUE ∨
          THR2_dispatch(THR2)=TRUE
 grd5  :  PR_waitingThread(process1)=FALSE
THEN
 act1  :  PR_waitingThread(process1)≔TRUE
END

mode_transition3
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1  :  prm1∈dom(MT_Smode>{PR_currentMode(process1)})
 grd2  :  MT_Response(prm1)=PLANNED
 grd3  :  in1_state(in1)=TRUE
 grd4  :  THR1_dispatch(THR1)=FALSE
          ∨ THR2_dispatch(THR2)=FALSE
THEN
 act1  :  PR_waitingThread(process1)≔FALSE
 act2  :  PR_SOM(process1)≔TRUE
END

mode_transition4
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1  :  prm1∈dom(MT_Smode>{PR_currentMode(process1)})
 grd2  :  PR_SOM(process1)=TRUE
 grd3  :  in1_state(in1)=TRUE
 grd4  :  THR1_completeACT(THR2)=TRUE
          ∨ THR2_completeDEC(THR1)=FALSE
THEN
 act1  :  PR_waitingThread(process1)≔FALSE
 act2  :  PR_SOM(process1)≔FALSE
 act3  :  PR_currentMode(process1)≔MT_Dmode(prm1)
 act4  :  PR_completeTRNS(process1)≔TRUE
END
```

FIGURE 10. Process mode transition.

b: TRANSFORMATION OF CONCRETE AADL CLASSES

The two classes in the concrete AADL classes (*BHA_annex* and *BHA_transition*) are mapped as the mapping approach of abstract AADL classes. The classes are mapped to the *Behavior_annex* context constants. The AADL class constant and association attributes are mapped to the *Behavior_annex* context constants. The AADL class variables are mapped to the *Behavior_annex* machine variables.

Algorithm 4 AC_connection *AC_connect* and Suprogram *call* Methods Description

loop
  if client thread is dispatched **then**
    if *clientThread_c* equal one **then**
      send call to the requires subprogram access by updating the value of *SQ_flag* to *TRUE*
      update the *clientThread_state* to *SUSPEND*
    end if
  end if
  if *SQ_flag* is *TRUE* **then**
    update corresponding buffer *SQ_flag* to *TRUE*
  end if
  if buffer *SQ_flag* is *TRUE* **then**
    update corresponding *SV_flag* to *TRUE*
  end if
  if *SV_flag* is *TRUE* **then**
    if *serverThread_state* is *AWAITING_DISPATCH* **then**
      update serverThread_state to *EXECUTION*
      call the subprogram by updating *SUB_flag* to *TRUE*
    end if
  end if
  if *SUB_flag* is *TRUE* **then**
    update *SUB_state* to *EXECUTION*
  end if
  while *SUB_time* is less than *SUB_EXtime* **do**
    increment *SUB_time* by one
  end while
  if *SUB_time* equal *SUB_EXtime* **then**
    upadate *SUB_Callflag* to *FALSE*
    reset *SUB_time* to zero
  end if
  if subprogram complete execution **then**
    update the *clientThread_state* to *EXECUTION*
    update the *ServerThread_state* to *AWAITING_DISPATCH*
  end if
end loop

The *state_transit* method is mapped to three machine events: *transitionName_Dispatch*, *transitionName_execute* and *transitionName_complete*, where,

- The *transitionName_Dispatch* event *refines* *threadName_start_dispatch* and has the same guard as the transition's guard
- The *transitionName_execute* event *refines* the *threadName_execute* and represents the delay of transition execution.
- The *transitionName_completeDispatch* event *refines* the *threadName_complete_dispatch*, has the same action as the state transition action, and subsequently changes *threadName_BHAstate* to the destination state.

AADL remote subprogram call

```
process process1
end process1;

process implementation process1.imp
    subcomponents
        T1: thread THR1.imp;
        T2: thread THR2.imp;
    connections
        CON1: subprogram access T2.access2 -> T1.access1;
end process1.imp;

thread THR1
    features
    access1: requires subprogram access SUB;
end THR1;

thread implementation THR1.imp
end THR1.imp;

thread THR2
  features
    access2 :provides subprogram access SUB;
end THR2;

thread implementation THR2.imp
    subcomponents
        SUB1: subprogram SUB;
    connections
        con: subprogram access SUB1 -> access2;
end THR2.imp;
subprogram SUB
end SUB;
```

The corresponding Event-B events

```
REQ1_call   ≙
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1   :   THR1_dispatch(THR1)=TRUE
 grd2   :   THR1_state(THR1)=EXECUTION
 grd3   :   THR1_c(THR1)=1
 grd4   :   SQ_flag(REQ1)=FALSE
THEN
 act1   :   SQ_flag(REQ1):=TRUE
 act2   :   THR_state(THR1):=SUSPEND
END

access1_connection   ≙
STATUS
 ordinary
ANY
 prm1
 prm2
WHERE
 grd1   :   prm1∈dom(AC_requirese>{REQ1})
 grd2   :   SQ_flag(REQ1)=TRUE
 grd3   :   prm2=AC_buffer(prm1)
THEN
 act1   :   SQ_flag(prm2):=TRUE
END

PRV1_call   ≙
STATUS
 ordinary
ANY
 prm1
WHERE
 grd1   :   prm1∈dom(AC_provide>{PRV1})
 grd2   :   prm2=AC_buffer(prm1)
 grd3   :   SQ_flag(prm2)=TRUE
 grd4   :   THR2_state(THR2)=AWAITING_DISPATCH
 grd5   :   THR2_dispatch(THR2)=FALSE
THEN
 act1   :   SV_flag(PRV1):=TRUE
 act2   :   SQ_flag(prm2):=FALSE
END
```

FIGURE 11. Remote subprogram call.

For example, Figure 12 shows the transformation of the AADL thread's behavior annex example.

c: BEHAVIOR_ANNEX MACHINE DECOMPOSITION

To allow the concurrent execution of the AADL threads, the *Behavior_annex* machine is decomposed. The functionality of the *Behavior_annex* MACHINE is separated so that

AADL thread along with its behavior annex

```
thread EXP
    features
        in1: in event port;
  end EXP;
  thread implementation Exp.imp
    annex behaviour_specification {**
    states
        s0:intitial complete final state;
        s1:complete state;
    transitions
    t0:s0[on dispatch in1]  s1
    **};
  end Exp.imp;
```

The corresponding Event-B events

```
EXP_BHAinitialization   ≙
STATUS
 ordinary
REFINES
 EXP_initialization
WHEN
 grd1   :   EXP_completeINI(EXP)=FALSE
 grd2   :   EXP_BHAstate(EXP)={S0↦INITIAL}
 grd3   :   PR_currentMODE(process1) ∈ THR_INmode(EXP)
 grd4   :   PR_loaded(process1)=TRUE
THEN
 act1   :   EXP_BHAstate(EXP):={S0↦AWAITING_DISPATCH}
 act2   :   EXP_completeINI(EXP):=TRUE
 act3   :   EXP_active(EXP):=TRUE
 act4   :   in1_state(in1):=FALSE
END

EXP_t0_dispatch   ≙
STATUS
 ordinary
REFINES
 EXP_dispatch
WHEN
 grd1   :   in1_state(in1)=TRUE
 grd2   :   EXP_BHAstate(EXP)={S0↦AWAITING_DISPATCH}
 grd3   :   EXP_dispatch(EXP)=FALSE
 grd4   :   EXP_c(EXP)=0
THEN
 act1   :   EXP_dispatch(EXP):=TRUE
 act2   :   EXP_c(EXP):=EXP_c(EXP)+1
 act3   :   EXP_state(EXP):=EXECUTION
END
EXP_t0_execution   ≙
STATUS
 ordinary
REFINES
 EXP_execution
WHEN
 grd1   :   EXP_dispatch(EXP)=TRUE
 grd2   :   EXP_c(EXP)<THR_EXtime(EXP)
 grd3   :   EXP_state(EXP)=EXECUTION
THEN
 act1   :   EXP_c(EXP):=EXP_c(EXP)+1
END
EXP_t0_completeDispatch   ≙
STATUS
 ordinary
REFINES
 EXP_complete_dispatch
WHEN
 grd1   :   EXP_c(EXP)=THR_EXtime(EXP)
 grd2   :   EXP_dispatch(EXP)=TRUE
 grd3   :   EXP_state(EXP)=EXECUTION
THEN
 act1   :   EXP_BHAstate(EXP):={S1↦AWAITING_DISPATCH}
 act2   :   EXP_c(EXP):=0
 act3   :   in1_state(in1):=FALSE
END
```

FIGURE 12. Thread behavior annex.

```
data Data1

end Data1;
process process1
    features
        PR_in1: in event port;
end process1;
process implementation process1.i
    subcomponents
        TH1: thread THR1.i;
         TH2: thread THR2.i;
    connections
        CON1: subprogram access TH1.RE1->TH2.PR1;
        CON2: port PR_in1 -> TH1.in1;
end process1.i;
thread THR1
    features
        in1: in event port;
        RE1: requires subprogram access SUB1;
        access1: requires data access Data1;
end THR1;
thread implementation THR1.i
    properties
        Dispatch_Protocol =>aperiodic;
    annex behaviour_specification{**
            states
            s0: intitial complete final state;
            s1: complete state;
            transitions
            s0[on dispatch in1]s1;
            s1[on dispatch in1]s0;
    **} ;
end THR1.i;
thread THR2
    features
        PR1: provides subprogram access SUB1;
        access2: requires data access Data1;
        properties
        Dispatch_Protocol =>aperiodic;
end THR2;
thread implementation THR2.i

    Subcomponents
        SUB: subprogram SUB1;
    annex behaviour_specification{**
            states
            s0: intitial state;
            s1: complete final state;
            transitions
            s0[on dispatch PR1]s1;
            s1[on dispatch PR1]s1;
            **} ;
end THR2.i;
Subprogram SUB1
end SUB1;
```

FIGURE 13. AADL example.

TABLE 3. Separation of events and variables.

	Process1 MACHINE	Thread1 MACHINE	Thread2 MACHINE
Internal variables	PR_in_variable PR_in_state	in_variable in_state THR1_active THR1_c	Out_variable Out_state THR2_active THR2_c THR1_t
Internal events	All process events	All THR1 events	All THR2 events
External variables	CON_buffer PR_currentMODE PR_loaded PR_stopped PR_SOM THR1_state THR2_state THR1_dispach THR2_dipatch THR1_completeINT THR2_completeINT THR1_completeACT THR2_completeACT THR1_completeDEC THR2_completeDEC	CON1_buffer PR_currentMODE PR_loaded PR_stopped PR_SOM THR1_state THR1_dispach THR1_completeINT THR1_completeACT THR1_completeDEC Data1_lock Data1_x SQ_flg(AC_buffer)	PR_currentMODE PR_loaded PR_stopped PR_SOM THR2_state THR2_dispach THR2_completeINT THR2_completeACT THR2_completeDEC Data1_lock Data1_x SQ_flg(AC_buffer)
External events	THR1_inilization1 THR1_inilization2 THR2_inilization1 THR2_inilization2 THR1_activation THR1_deactivation THR2_activation THR2_deactivation THR1_finlization THR2_finlization	Process1_complete_load Process1_stop Mode_transiton1 Mode_transiton2 Mode_transiton3 Mode_transiton4 THR2_getresource THR2_releaseesource PR_sendoutput	Process1_complete_load Process1_stop Mode_transiton1 Mode_transiton2 Mode_transiton3 Mode_transiton4 THR1_getresource THR1_releaseesource AC_connection

the process and each thread constitute a sub-model. The partitioning of the example in Figure 13 is shown in Table 3. The *Process1* MACHINE shares all external variables with the *THR1* and *THR2* machines and communicates with the tow thread machines via the external events. The *THR1* also shares the access connection buffer relative variables with the *THR2* MACHINE.

IV. VERIFICATION OF THE TRANSFORMATION CORRECTNESS

To verify the correctness of the model transformation, it is essential to verify the preservation of the AADL semantics and verify that the mapping rules are deterministic and terminating.

We have used the UML class diagram as the intermediary between the AADL and the Event-B model. Therefore, first, we start to verify the correctness of the class diagram description. Then, we verify the correctness of the AADL class diagram transformation into Event-B. To verify the semantics preservation of our transformation approach, we propose a graph homomorphism [23], [24], which verifies

the structural relationships between two elements of two different models. Therefore, it presents the semantic preservation of the mapping from the source to the target model and preserves the consistency of the transformation approach.

Definition 1: The two graphs G and H with
$N(G)$: set of nodes of the graph G,
$N(H)$: set of nodes of the graph H,
$E(G)$: set of edges of the graph G,
$E(H)$: set of edges of the graph H,
The function mapping f is from graph G to graph H, where $f: G \to H$ such that:

$$\forall x, y \in N(G) \wedge f(x), f(y) \in N(H) \wedge xy \in E(G) \Rightarrow f(x)f(y) \in E(H)$$

then f is called graph homomorphic.

The mapping rules are deterministic when they always get a unique result with the same input.

Definition 2: The function mapping f is from graph G to graph H, where $f: G \to H$ such that:

$$\forall x, y \in N(G) \wedge f(x) = f(y) \Rightarrow x = y$$

then f is called deterministic

To verify the semantics preservation and determinism, we propose a graph homomorphism of the description of model's elements between the AADL and UML class diagram and then the mapping between the AADL class diagram and Event-B. Two graphs function mappings are defined, a and b, where a is a mapping function between two model graphs, AADL*(AD)* and UML class diagram*(CL)*, and b is a mapping function between two models, AADL class diagram*(CL)* and Event-B*(EB)*, so we must prove that a and b are graph homomorphic and deterministic.

In general, our approach was proposed based on the perspective of grammatical structure and semantics of the source and target model. Because both sides of the transformation model have exactly no ambiguous grammatical structure, we have applied a one-to-one mapping between the source and the target model.

According to the AADL grammatical structure described by the context-free grammar, we have combed out the transformation elements of the AADL model and the constituent relationships among the elements. Then, we have applied a one-to-one description of the AADL model elements, behavior of the elements, and relationship among the elements to the UML class diagram. Each description corresponds to a determined and unique description result. Therefore, in the mapping function a, we have ensured that:

$$\forall x, y \in N(AD) \wedge f(x), f(y) \in N(CL) \wedge xy \in E(AD) \Rightarrow f(x)f(y) \in E(CL)$$

Hence, a is graph homomorphic.

We also have ensured that:

$$\forall x, y \in N(AD) \wedge f(x) = f(y) \Rightarrow x = y$$

Hence, a is deterministic.

The one-to-one mapping started from the top leaf in the AADL structured composition, and the termination condition of the transformation is the atomic element in the AADL model, which is the lowest leaf node in the AADL structured composition. The number of nodes decreases in each transformation step. Hence the transformation is terminating [25].

Similarly, it has been proven that the function mapping b is graph homomorphic, deterministic and terminating.

V. CASE STUDY

A. DESCRIPTION OF THE CTCS-3 MA SYSTEM

To show the efficacy of our approach in the verification of the AADL model, this section proposes the formal verification experimental results for the Movement Authority (MA) control of the Chinese Train Control System. The authors in [26] present the analysis and description of how the train controls and monitors its velocity. The MA system consists of three basic components: *i) the train* periodically sends its current state (every 200 milliseconds) to the controller and receives the computed acceleration from the controller. *ii) The Radio Block Center (RBC)* provides and extends the MAs to the trains according to the information received from the trackside and on-board controller. *iii) The On-board controller* controls the train velocity by modifying its acceleration. The MA package consists of a set of segments, length and endpoint (EoA). Each segment contains velocity limitations v1 and v2 (v1 $\leqslant$ v2) and segment endpoint (e). The train requests for new movement's authority as it arrives at a specific distance (SR) from EoA. In this case study, we assume the length of the MA is 8 kilometers, the length of the SR is 1.5 kilometers, and all segments have identical length and speed limits. The train starts with a speed of 0 m/s.

B. AADL MODEL OF THE CTCS-3 MA SYSTEM

We build the hybrid CTCS-3 AADL model based on the description introduced in [26]. Figure 14 illustrates the AADL model of CTCS-3 MA, which comprises the *acceleration_control* process with three modes: *STOP, READY* and *MOVING*. The *acceleration_control* process contains five threads, (*Get_trainINFO, Train_start, Get_MA, Calculate_distance* and *Check_acceleration*) and three data subcomponents (*MA, Segment*, and *Train_INFO*). The *STOP* is the initial mode; once the process has received the event on the *train_ON* port, the *Train_start* thread is activated, the current mode is changed to *READY*, and *Train_start* applies for a new MA. Once the train has received the new MA, *Get_trainINFO, Get_MA, Calculate_distance* and *Check_acceleration* are activated, *Train_start* is deactivated, and the current mode is changed to *MOVING*. The *Get_MA* receives the new MA and sends an event to *Check_acceleration*, which computes and sends a new acceleration. The *Calculate_distance* thread applies for a new MA as train position $\leqslant$ EoA-SR. If the *Get_MA* thread does not receive a new MA, all process threads are immediately deactivated, and the current mode is changed to *STOP*.

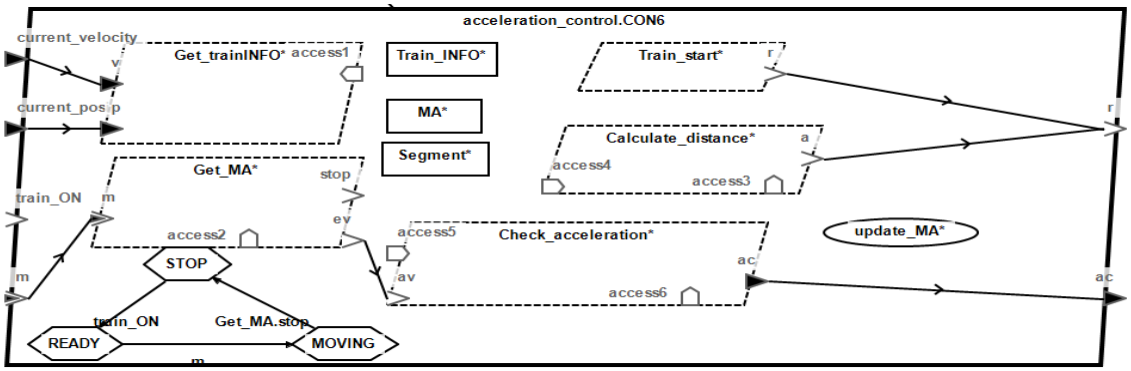

FIGURE 14. AADL CTCS-3 MA system.

C. FORMAL VERIFICATION

We transform the AADL CTCS-3 MA model to Event-B according to the above defined mapping rules. In this section, we define the properties that are verified as theorems or invariants. The RODIN platform automatically generates and proves the corresponding proof obligation rules for each theorem and invariant. Some of the generated rules require an external prover such as Atelier B prover

1) **Safety properties** are described as ''something bad never happens''

- System safety constraints are defined as invariants to ensure that the system safety constraints are preserved by each event. The CTCS-3 MA system has two safety constraints: **(i) the train is constantly moving forward**, otherwise it has already stopped. This constraint is defined as the following invariant:

moving_forward: current_velocity (Train_INFO1) $\geqslant 0$

where current_velocity indicates the current train velocity. For this invariant, the PO rule **event/ moving_forward /INV** is automatically generated for each event. **(ii) The train must send a request for a new movement authority** as it reaches a specific distance (SR) from the (EoA) of MA. This constraint is defined as the following invariant:

MA_extention: current_position (Train_INFO1) $\geqslant$ MA_EoA(MA1)-SR

where *current_position* indicates the current train position that has been saved in the *Train_INFO1* shared data resource. For this invariant, the PO rule **event/ MA_extention /INV** is automatically generated for each event.

- **Timing correctness** represents the state that the thread execution time never exceeds the thread maximum execution time. This constraint is defined as the following invariant:

timing_correctness: threadName_c(threadName) $\leqslant$ THR_Extime (threadName)

For example, in the thread Get_MA, the timing correctness invariant is:

MA_timing_correctness: Get_MA_c(Get_MA) $\leqslant$ THR_Extime (Get_MA)

For this invariant, the PO rule **event/MA_timing_c-orrectness /INV** is automatically generated for each event.

- **Reachability** refers to the ability of transition from one state to another with one or more events. The AADL model requires the verification of mode, and the behavior annex state reachability **(i) mode reachability** refers to the ability of transition from a mode to another while receiving a new event/event data on one of event/event data ports. This condition is defined as the following theorem:

Mode_reachability: $\forall$ a.a $\in$ modes $\wedge$ a $\in$ ran(PR_modes) $\Rightarrow$ a $\in$ ran (MT_Smode)

The Mode_reachability theorem implies that each mode that belongs to process modes has an outgoing transition to another mode. The generated PO rule for this theorem is **Mode_reachability/ THM**.

(ii) Behavior states reachability refers to the ability of transition from a behavior state to another with one or more guards. We define this condition as the following theorem:

State_reachability: $\forall$ a.a $\in$ state $\wedge$ a $\in$ ran(BHA_states) $\Rightarrow$ a $\in$ ran (TRNS_Sstate)

The State_reachability theorem implies that each state belonging to a thread behavior annex state has an outgoing transition to another state. The generated PO rule for this theorem is **State_reachability/ THM**.

- **Deterministic** refers to the conditions where there are no two or more outgoing transitions, which lead to different states with the same events. For example, the following theorem is defined to verify whether the process modes are deterministic:

Mode_ deterministic: ∀ x, y. x∈ Mode_transition
∧ y ∈ Mode_transition ∧ x ≠ y
∧ MT_Smode(x) =MT_Smode(y)
MT_Dmode(x) ≠ MT_Dmode(y)
⇒
MT_DISport(x) ≠ MT_DISport(y)

The generated PO rule for this theorem is **Mode_ deterministic / THM**.

- **Deadlock-free**: in concurrent computing, deadlock refers to two conditions **(i) First**, each member in a group is waiting for some other members to take action; therefore, they cannot interact with the environment. In the AADL model, if the port connection between two threads is deleted, the destination thread cannot be dispatched. Hence, the destination thread will wait to receive a new event or event data. This condition is defined as the following theorem:

Deadlock_free1: ∀ a.a ∈ ℙ (DIS_ports) ∧ a ∈
ran (THR_DISports) ⇒
a ∈ran (CON_Dport)

The Deadlock_free1 theorem implies that each thread dispatch port must be connected to any other corresponding thread port. The generated PO rule for this theorem is **Deadlock_free1/THM**.

(ii) The second condition refers to the state that two members in a group are sharing the same two resources and waiting for each other to release the resources. In AADL, this condition refers to the state where two threads are sharing two different data components and waiting for each other to release them. Figure 15 shows the CTCS-3 MA system model deadlock state, where the *Check_acceleration* and *Calculate_distance* threads are locking the two data components *MA* and *Train_INFO* and waiting for each other causing a deadlock.

This condition is defined for the two threads as follows:

Deadlock_free2: ((MA_lock (MA1) = TRUE
∧ MA_accessingThread (MA1) = Check_acceleration)
∧ (Train_INFO_lock (Train_INFO1)= TRUE
∧ Train_INFO_accessingThread (Train_INFO1)
= Calculate_distance))
⇒
Check_acceleration ∉ Train_INFO_accessQueue
(Train_INFO1)
∧ Calculate_distance∉ MA_accessQueue (MA1)

This theorem implies that if the *Calculate_distance* and *Check_acceleration* threads have a requires

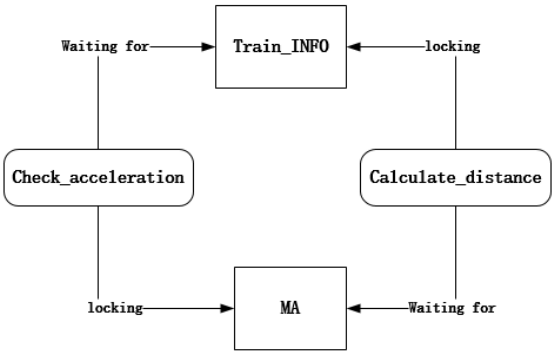

FIGURE 15. Two-thread deadlock condition.

data access feature to tow different data components, *MA* and *Trian_INFO*, the *Check_ acceleration* is locking *MA* and wants to access *Train_INFO*. Simultaneously, *Calculate_distance* is locking the *Train_INFO* and wants to access the *MA*. The *Check_acceleration* must not be in the *Train_INFO* waiting queue. The *Calculate_distance* must not be in the *MA* waiting queue. The generated PO rule for this theorem is **Deadlock2/ THM**.

2) **Liveness properties** refer to the state of "something good eventually happens". The liveness properties are always expressed with linear-time temporal logic (LTL) formulas, which is a modal temporal logic with modalities refers to time. We propose a set of proof rules to verify the progress properties [27]: something must eventually occur if some conditions hold (□ (P1 ⇒ ◇ P2)); we verify the properties of (□(thread dispatch condition ⇒ ◇ thread dispatch)) and (□(mode switching ⇒ ◇; all new mode's threads are active, and those do not belong to new mode are not active)). All liveness properties are described as invariants. For example, the following invariant is defined for Calculate_distance thread:

CalculateD_liveness1: CalculateD_t (Calculate_distance)
= THR_period (Calculate_distance)
⇒
CalculateD_dispatch
(Calculate_distance) = TRUE

The generated PO rule for this invariant is **event/ CalculateD_liveness1/INV**.

3) **The trace refinement properties** refer to the state where the concrete model satisfies the abstract one. In Event-B, the RODIN platform automatically generates the proof obligation rules for concrete events in the *Behavior_annex MACHINE*. Two PO rules are generated **(i) simulation (SIM)** to ensure that when a concrete event is executed and performs actions, what it executes does not contradict what the corresponding abstract event executes. **(ii) The invariant**

preservation (INV) verifies that each abstract invariant is preserved by both concrete and abstract events. For example, the *Check_acceleration* thread has a behavior annex. For the corresponding behavior annex events of each state transition, two types of PO are generated (SIM and INV) to ensure that the *Behavior_annex MACHINE* events satisfy the refined *MACHINE1* events and invariants.

VI. RELATED WORK

Several studies on the formal verification of AADL using model transformation have been presented.

D'Souza *et al.* [30] use the Event-B refinements and decompositions to capture the semantics of the AADL model defined through successive refinements. Using the transformation of AADL components to Event-B presents the ability to formally prove architectural requirements related to the correctness of AADL models. Their work only focuses on the AADL architectural semantics, and the transformation of AADL behavioral semantics is not included. Therefore, many properties related to behavioral semantics were not verified. Generally, we have addressed different issues and presented different paths, including the general mapping rules of AADL components and mapping behavior of the AADL components to Event-B, whereas they only present the transformation of the AADL subset in the case study that they used. For the refinement correctness verification, the above paper addresses the correctness of the continuous refinement of the AADL model, whereas we have addressed the issues of the semantic reliability of the AADL model at each refinement level.

Chkouri *et al.* [9] transform AADL to BIP (Behavior Interaction Priority), which is a framework to model real-time components. This transformation provides the simulation of AADL models and formal verification techniques such as model-checking (Aldebaran and observers tools).

Berthomieu *et al.* [7] propose a formal verification for AADL with its behavioral annex using a high-level view tool. Relying on Fiacre, the Tina model is generated from the AADL model, and the verification activities are presented. Bodeveix *et al.* [8] express the semantics of the AADL and FIACRE subsets in a common framework, which is called the timed transition systems (TTS).

Mkaouar et al [12] introduce the transformation of an interesting subset of the AADL model to an LNT [13]. which is supported by the CADP toolbox.

Ölveczky *et al.* [10], [11] present a formal object-based semantics for the AADL subset. The generated semantics is executed in Real-Time Maude. They propose an AADL LTL model checking with the OSATE integrated tool called AADL2Maude.

Yang *et al.* [14] transform AADL to Timed Abstract State Machines (TASM) with machine semantic-preservation and verify the semantic preservation of the transformation rules by a theorem prover (Coq). Hu *et al.* [15] propose a translation of AADL to TSAM and verify some properties (deadlock and reachability) using UPPAAL.

Bao *et al.* [31] present the Uncertainty annex, which is a proposed extension language to AADL. They transform the uncertain-aware Hybrid AADL to NPTA and present a verification based on the UPPAAL-SMC. Johnsen *et al.* [28] use semantic anchoring to present the transformation rule of semantics of the AADL subset to timed automata constructs, which is an input language to the UPPAAL model-checker.

Yang *et al.* [4] construct formal semantics of AADL using machine-reachable CSP, analyze and verify the deadlock and livelock using the tool FDR. Ahmad *et al.* [5] propose the formal semantics of the synchronous subset of AADL by using HCSP and verify the correctness of the AADL model by an in-house developed theorem prover, the Hybrid Hoare Logic (HHL) prover. Zhang *et al.* [6] introduce a set of transformation rules for AADL to the stateful timed CSP and verify the critical behavior properties of the AADL model by PAT.

Yu *et al.* [32] translate the AADL model into SIGNAL models in order to avoid AADL semantics ambiguities. This translation provides formal analysis and represents a bridge between AADL and SYNDEX. The SYNDEX model is used to do distribution, scheduling and architecture exploration. Gautier *et al.* [29] present a formal model of automata based on clock relations, this model is called polychronous automata. Then they refine the AADL transformation into polychronous models which are introduced in [32] by presenting the AADL Behavior annex as polychronous automata. The polychronous automata model provides the verification and analysis of properties such as deadlock-freeness and schedulability.

Table 4 shows the differences between our work and previous works, where the comparison is determined according to the following principles:

First, the AADL subset is included in the work. The previous works only focused on a small AADL subset. With the development of real-time systems, they become larger and more complex. Therefore, when it has been modeled using AADL, the AADL components included in the model are larger. Our work focuses on extending the AADL subset. We have compared the included subset by comparing the AADL components, connections, Shared variables, Behavior annex, and subprogram call. For the AADL connection, our work includes five types of connections, three port types (data, event, and event data), parameter, and access connections. The ''+'' sign in Table 4 refers to each connection type. The Behavior annex is not included in some works, and others include only a small Behavior annex subset. Our work includes the entire Behavior annex subset (state, variables, state transitions), which is denoted by ''+'' in Table 4. Our work includes two types of subprogram call: local and remote calls. Each subprogram call is denoted by ''+'' in Table 4, whereas the previous works do not include the remote subprogram call. The shared variables that no previous work considered are included in our work.

TABLE 4. Comparison of related work.

Language	Verification tool	Support subset					Verified properties		
		Components	Behavior annex	Shared variable	Connection	Subprogram call	Shared data deadlock	Trace refinement	LTL
BIP [9]	BIP Framework	++++	++	-	+++	-	-	-	✓
LNT [13]	CADP	+++	-	-	+++	-	-	-	✓
FIACRE [8]	Tina	+++	-	+	+++	-	-	-	✓
Maude [10] [11]	Maude	++	+++	-	+++	-	-	-	✓
UPPAAL Time automata [28]	UPPAAL	++	-	-	+	-	-	-	-
TSAM [15]	UPPAAL	++	+++	-	+++	-	-	-	✓
CSP [4]	FDR	++++	-	+	++	++	-	-	-
Stateful timed CSP [6]	PAT	+++	+++	-	++++	+	-	✓	✓
SIGNAL and Polychronous automata [29]	Polychronous automata	++	++	-	+++	-	-	-	-
Event-B (our work)	**Proof obligation**	**++++**	**+++**	**+**	**+++++**	**++**	✓	✓	✓

Second: the verified properties. As shown in Table 4, we have presented the verification of the additional properties, which were not verified by previous work such as the shared data deadlock. These properties must be verified.

Third: the transformation correctness. Most related works do not consider the transformation correctness, which is presented in our work.

VII. DISCUSSION

The main purpose of this paper is to present a formal verification of the critical properties of the AADL models by Event-B.

Our results show that the transformed model is extendable; as we have used the UML class diagram as a role intermediary, more AADL subsets can be added as new classes and connect with the related classes using class relationships.

Moreover, the Event-B with its refinements and extension relationships makes the transformation of the new AADL classes easy without retransforming the entire model. Event-B also provides verification that the properties in the abstract level are preserved through refinement.

The Event-B proof obligation is effective to verify the AADL critical properties, so any properties can be expressed by using invariants and theorems. The critical properties have been automatically verified using RODIN auto-prover, and some properties require external prover such as Atelier B prover. Using the RODIN Prob tool, the behavior of the critical properties has been observed and experimented.

Compared to previous works, our work has satisfied all requirements of the current real-time systems. We have attempted to make our approach more effective to verify most of the current systems, whose design has become more complex and contains a larger AADL subset. Unlike previous methods, by using the Event-B refinement, the AADL subset can be transformed gradually to ensure the traceability of Event-B against the AADL.

Our mapping rules are clear and have no ambiguous semantics. The mapping rules have included all details of the transformation of AADL to the Event-B model, which makes the implementation of the automatic transformation tool easy according to these mapping rules. The transformation tool can be a Plug-in unit of Osate with RODIN.

In order to ensure the correctness of the transformation, we have ensured that the transformation is deterministic, terminating and preserves the semantic. We have first ensured the correctness of the class diagram description of AADL, then ensured the correctness of the mapping of the AADL class diagram into Event-B. The AADL class diagram description has been defined from the perspective of the grammatical structure since both sides of the transformation model have exactly no ambiguous grammatical structure. We used Osate tool to further clarify the AADL semantics through experimental verification such as the concept of flow in AADL and the behavioral semantics of threads. Then, we applied a one-to-one description of the AADL model elements, the behavior of the elements, and the relationship among the elements to the UML class diagram. Each AADL element has corresponding, determined and unique description result. The AADL class diagram has been transformed into Event-B by one-to-one mapping since each AADL class diagram element, attribute, relationship and method has a unique transformation result.

VIII. CONCLUSION

The formal verification of AADL models using the model transformation aspect to a formal language has been used to verify some critical safety properties. This paper presented the formal verification of the AADL model by transforming the semantics of the model into Event-B. The Event-B has been selected because it can cover all AADL architectural and behavioral semantics with preservation of the AADL properties. It can also formally verify most of AADL critical safety and liveness properties.

We have introduced the AADL model subset description using the UML class diagram, which has played the role of intermediary for the transformation. By modeling the UML

class diagram for AADL, we have described the details, features, and behavior of each AADL component and the relationships of the components. We have defined a set of mapping rules from the AADL class diagram to Event-B; these mapping rules were presented according to the one-to-one mapping of the AADL classes, attributes, methods and the relationships among the AADL classes into the Event-B model. The behaviors of the AADL components, which are described as class methods in the AADL class diagram, were mapped to Event-B based on algorithms that were defined to represent the detailed semantics of the component behavior, such as the description of thread dispatch. These algorithms have been translated to Event-B semantics and validated using the RODIN ProB tool.

This paper involves the transformation of most of the AADL components, which have been recently included in the real-time system model due to the expansion of these systems and their increasing complexity. We have considered a large AADL subset including software components, process, all dispatch thread types (periodic, aperiodic, sporadic, and timed), subprograms, data components, connections (data port, event port, event data port, parameters, access), remote and local subprogram calls, shared variables by several threads, mode transitions and thread behavior annex. After the AADL model has been transformed, the critical properties to be verified have been defined as Event-B theorems and invariants. The properties have been extracted from the behavior of the system that causes these properties to occur, and they have been mathematically expressed using Event-B invariant and theorems. Then, the corresponding proof obligation rules of these invariants and theorems have been automatically generated and verified by the RODIN platform. This paper has verified additional safety properties such as Shared data deadlock and proven the correctness of AADL refinements. Moreover, the transformation correctness has been verified using graph homomorphism.

The effectiveness of our approach has been demonstrated by modeling the AADL of the Movement Authority (MA) control of the Chinese Train Control System. The AADL model has been transformed to Event-B using our methodology; then, the transformed Event-B model has been verified using proof obligation rules.

REFERENCES

[1] *Architecture Analysis and Design Language (AADL)*, SAE Int., New York, NY, USA, 2017.

[2] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2010.

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: An open toolset for modelling and reasoning in event-B," *Int. J. Softw. Tools Technol. Transf.*, vol. 12, no. 6, pp. 447–466, Nov. 2010.

[4] C. Yang, Y. Dong, F. Zhang, E. Ahmad, and B. Gu, "Formal semantics of AADL models with machine-readable CSP," in *Proc. IEEE/ACIS 11th Int. Conf. Comput. Inf. Sci.*, May 2012, pp. 565–571.

[5] E. Ahmad, Y. Dong, S. Wang, N. Zhan, and L. Zou, "Adding formal meanings to AADL with hybrid annex," in *Proc. Int. Conf. Formal Aspects Compon. Softw.* Cham, Switzerland: Springer, 2014, pp. 228–247.

[6] F. Zhang, Y. Zhao, D. Ma, and W. Niu, "Formal verification of behavioral AADL models by stateful timed CSP," *IEEE Access*, vol. 5, pp. 27421–27438, 2017.

[7] B. Berthomieu, J.-P. Bodeveix, S. Dal Zilio, P. Dissaux, M. Filali, P. Gaufillet, S. Heim, and F. Vernadat, "Formal verification of AADL models with fiacre and tina," in *Proc. Eur. Congr. Embedded Real-Time Softw.*, 2010, pp. 1–9.

[8] J.-P. Bodeveix, M. Filali, M. Garnacho, R. Spadotti, and Z. Yang, "Towards a verified transformation from AADL to the formal component-based language FIACRE," *Sci. Comput. Program.*, vol. 106, pp. 30–53, Aug. 2015.

[9] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP-application to the verification of real-time systems," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.* Berlin, Germany: Springer, 2008, pp. 5–19.

[10] 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*. Berlin, Germany: Springer, 2010, pp. 47–62.

[11] P. C. Ölveczky, "Formal model engineering for embedded systems using real-time Maude," 2011, *arXiv:1107.0063*. [Online]. Available: http://arxiv.org/abs/1107.0063

[12] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, "From AADL model to LNT specification," in *Proc. Ada-Eur. Int. Conf. Reliable Softw. Technol.* Cham, Switzerland: Springer, 2015, pp. 146–161.

[13] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, "A formal approach to AADL model-based software engineering," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 2, pp. 219–247, Apr. 2020.

[14] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin, "From AADL to timed abstract state machines: A verified model transformation," *J. Syst. Softw.*, vol. 93, pp. 42–68, Jul. 2014.

[15] K. Hu, T. Zhang, Z. Yang, and W.-T. Tsai, "Exploring AADL verification tool through model transformation," *J. Syst. Archit.*, vol. 61, nos. 3–4, pp. 141–156, Mar. 2015.

[16] J.-R. Abrial, "A system development process with event-B and the rodin platform," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2007, pp. 1–3.

[17] R. Silva, C. Pascal, T. S. Hoang, and M. Butler, "Decomposition tool for event-B," *Softw., Pract. Exper.*, vol. 41, no. 2, pp. 199–208, Feb. 2011.

[18] M. Butler, "Decomposition structures for event-B," in *Proc. Int. Conf. Integr. Formal Methods*. Berlin, Germany: Springer, 2009, pp. 20–38.

[19] P. Boström, F. Degerlund, K. Sere, and M. Waldén, "Concurrent scheduling of event-B models," 2011, *arXiv:1106.4100*. [Online]. Available: http://arxiv.org/abs/1106.4100

[20] T. S. Hoang and J.-R. Abrial, "Event-B decomposition for parallel programs," in *Proc. Int. Conf. Abstract State Mach., Alloy, B Z.* Berlin, Germany: Springer, 2010, pp. 319–333.

[21] S. Hallerstede, "On the purpose of event-B proof obligations," *Formal Aspects Comput.*, vol. 23, no. 1, pp. 133–150, Jan. 2011.

[22] M. Leuschel and M. Butler, "ProB: A model checker for B," in *Proc. Int. Symp. Formal Methods Eur.* Berlin, Germany: Springer, 2003, pp. 855–874.

[23] H. Cao, S. Ying, and D. Du, "Towards model-based verification of BPEL with model checking," in *Proc. 6th IEEE Int. Conf. Comput. Inf. Technol. (CIT)*, Sep. 2006, p. 190.

[24] A. B. Younes, Y. B. Hlaoui, and L. J. B. Ayed, "A meta-model transformation from UML activity diagrams to event-B models," in *Proc. IEEE 38th Int. Comput. Softw. Appl. Conf. Workshops*, Jul. 2014, pp. 740–745.

[25] H. S. Bruggink, B. König, and H. Zantema, "Termination analysis for graph transformation systems," in *Proc. IFIP Int. Conf. Theor. Comput. Sci.* Berlin, Germany: Springer, 2014, pp. 179–194.

[26] E. Ahmad, Y. Dong, B. Larson, J. Lü, T. Tang, and N. Zhan, "Behavior modeling and verification of movement authority scenario of Chinese train control system using AADL," *Sci. China Inf. Sci.*, vol. 58, no. 11, pp. 1–20, Nov. 2015.

[27] T. S. Hoang and J.-R. Abrial, "Reasoning about liveness properties in event-B," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2011, pp. 456–471.

[28] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated verification of AADL-specifications using UPPAAL," in *Proc. IEEE 14th Int. Symp. High-Assurance Syst. Eng.*, Oct. 2012, pp. 130–138.

[29] T. Gautier, C. Guy, A. Honorat, P. Le Guernic, J.-P. Talpin, and L. Besnard, "Polychronous automata and their use for formal validation of AADL models," *Frontiers Comput. Sci.*, vol. 13, no. 4, pp. 677–697, Aug. 2019.

[30] M. D'Souza, S. Ramesh, and M. Satpathy, "Architectural semantics of AADL using event-B," in *Proc. Int. Conf. Contemp. Comput. Informat. (ICI)*, Nov. 2014, pp. 92–97.
[31] Y. Bao, M. Chen, Q. Zhu, T. Wei, F. Mallet, and T. Zhou, "Quantitative performance evaluation of uncertainty-aware hybrid AADL designs using statistical model checking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 12, pp. 1989–2002, Dec. 2017.
[32] H. Yu, Y. Ma, T. Gautier, L. Besnard, J.-P. Talpin, P. Le Guernic, and Y. Sorel, "Exploring system architectures in AADL via polychrony and SynDEx," *Frontiers Comput. Sci.*, vol. 7, no. 5, pp. 627–649, Oct. 2013.

ABEER SAEED ABDO HADAD received the B.Eng. degree in software engineering from Taiz University, Taiz, Yemen, in 2012. She is currently pursuing the M.Eng. degree in software engineering with Northwestern Polytechnical University, Xi'an, China. Her research interest includes formal verification of AADL models.

CHUNYAN MA received the Ph.D. degree in computer science and engineering from Northwestern Polytechnical University, in 2009. Since 2009, she has been working with the Department of Software Engineering, Northwestern Polytechnical University, where she is currently an Associate Professor. She has over 20 publications at major venues, including *Software Qual Journal*, COMPSAC, and QSIC. Her research interests include software analysis and testing and modeling. She has won research funding from several competitive sources, such as NSFC (Program No. 61103003) and Natural Science Basic Research Plan in Shaanxi Province of China (2011JQ8008).

ADEEB ABDULWAKEEL OBADI AHMED received the B.Eng. degree in industrial and manufacturing system engineering from Taiz University, Taiz, Yemen, in 2012. He is currently pursuing the M.Eng. degree in mechanical engineering with Xidian University, Xi'an, China. His research interest includes manufacturing systems.