A Mechanized Semantics for Component-based Systems in the HAMR AADL Runtime

Stefan Hallerstede[1] and John Hatcliff[2]

[1] Aarhus University
[2] Kansas State University, Manhattan KS 66506, USA

Abstract. Many visions for model-driven component-based development emphasize models as the "single source of truth" by which different forms of analysis, specification, verification, and code generation are integrated. Such a vision depends strongly on a clear modeling language semantics that provides different tools and stakeholders with a common understanding of a model's meaning. In this paper, we report on a mechanization of a formal semantics in the Isabelle theorem prover for key aspects of the SAE standard AADL modeling language. A primary goal of this semantics is to support component-oriented contract specification and verification as well as code generation implemented in the HAMR AADL model-driven development tool chain. We provide formal definitions of run-time system state, execution steps, reachable states, and property verification. Use of the mechanization for real-world applications is supported by automated HAMR translation from AADL models into the Isabelle specifications. In addition to general verification support, we define well-formedness properties and associated proofs for models, system states, and traces that are automatically proven for HAMR-generated Isabelle models.

1 Introduction

Model-driven development tools continue to gain traction for building and assuring safety-critical systems. The Architecture Analysis and Design Language (AADL) [1] stands out among other standardized modeling languages due to its stronger semantic interpretation of modeling elements. Stronger emphasis on semantics has led to AADL's use in numerous large-scale industrial research projects, particularly those prioritizing formal methods. An ecosystem of tooling has developed around AADL that includes analyses for behavior specification and verification, resource utilization and timing, trade space exploration, hazard analysis, code generation, and assurance case development.

Despite this emphasis on semantics, semantic descriptions in the current version of the standard are presently mostly in narrative form, or with limited use of timed automata that are not strongly integrated with other descriptions. This makes it more difficult to establish the soundness of model-based analyses and verification, the correctness of code generation, and a consistent semantic interpretation across multiple tools. Recently, Hugues has led an effort within

the AADL committee to develop a road map for integrating formal definitions of semantics into the standard. This is supported in part by an open source mechanization of the AADL static semantics and various supporting analyses in the Coq theorem prover [22].

In an accompanying coordinated effort, Hatcliff, Hugues and others [19] developed a rule-based specification of important elements of the AADL Run-Time Services (RTS). The AADL RTS aim to (a) hide the details of specific RTOS and communication substrates (e.g., middleware) and (b) provide AADL-aligned system implementations with canonical platform-independent actions that realize key steps in the integration and coordination of component application logic. Though it is designed to be implementation-independent, traceability to the RTS formalization in [19] has been emphasized in the High Assurance Modeling and Rapid engineering framework (HAMR) AADL code generation framework [17] as a guide in developing consistent code generation on multiple platforms and in designing integrated model and code behavior contract specification and verification [21] and property-based testing [18]. HAMR provides code generation in C and in Slang [26] (a safety-critical subset of Scala, that can also be translated to C) and system deployments can be generated for the JVM, Javascript, Linux, and the seL4 microkernel.

To improve the utility, feature coverage, and confidence in the formalization in [19], we are developing a suite of interconnected artifacts including an executable version of the semantics (to serve as an abstract reference implementation, e.g., to use in model-based testing) and an encoding of the semantics in the Isabelle theorem prover. One of HAMR's code generation targets is the seL4 micro-kernel whose semantics and correctness properties have been formally specified and proved in Isabelle – providing one of the most significant applications of formal methods to date [23]. One of the motivations for chosing Isabelle over another theorem prover is to provide semantic specifications for AADL and HAMR that can eventually be connected to the seL4 formalization.

In this paper, we report on the Isabelle-based mechanization of AADL semantics, inspired by the definitions in [19], as well as several significant novel extensions that provide foundations for verification and refinements of the framework to address characteristics of HAMR code generation target platforms. We refer to this mechanization as AADL-HSM (AADL HAMR Semantics Mechanization). The specific contributions of this paper are as follows.

- We provide a mechanization of the notions of port, thread, and system states as well as the AADL run-time services given in [19].
- We give definitions of AADL communication that accommodate AADL port-based communication properties and that can be instantiated to the communication semantics of different deployment platforms.
- We give definitions for thread scheduling that can accomodate, e.g. the static cyclic scheduling used by HAMR on recent industry projects [5].
- We introduce basic definitions of a property framework for reasoning about threads and system properties. This lays the formal foundation for address-

ing the rationale and soundness for the GUMBO contract framework for component verification [21] and testing [18].
– We extend the HAMR code generator to translate core AADL instance models and HAMR initial system states into the mechanization.

This paper summarizes the approach and key aspects of the above contributions, but the primary artifacts substantiating the results are our Isabelle specifications publicly available at [29]. These are formatted and commented using Isabelle's documentation framework to provide a 100+ page PDF readable guide to the semantics of the AADL subset supported by HAMR. The HAMR distribution is available at [28].

2 AADL Background

SAE International standard AS5506C [1] defines the AADL core language for expressing the structure of embedded, real-time systems via definitions of components, their interfaces, and their communication.

AADL provides a precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements using a component-based approach. Components have a category that defines a standard interpretation. Categories include software (e.g., threads, processes), hardware (e.g., processor, bus), and system (interacting hardware and software). Each category also has a distinct set of standardized properties that can be used to configure the specific component's semantics for various aspects: timing, resources, etc.

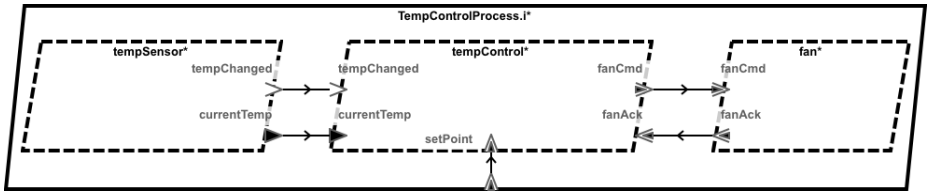

Fig. 1. Temperature Control Example (excerpts) – AADL Graphical View

Figure 1 presents a portion of the AADL standard graphical view for a simple thermostat that maintains a temperature according to a set point structure. The `system` (not shown) contains a `process` called `tempControlProcess`. This process consists of three threads: `tempSensor`, `tempControl`, and `fan`, shown in the figure from left to right. Passing data from one thread to another is done through the use of `ports`. Each port can be classified as an *event port* (e.g., to model interrupt signals or other notification-oriented messages without payloads), a *data port* (e.g. modeling shared memory between components or distributed memory services where an update to a distributed memory cell is automatically propagated to other components that declare access to the cell), or an *event data port* (e.g., to model asynchronous messages with payloads, such as in publish-subscribe frameworks). We refer to these classifications as

the port's *kind*. Inputs to event and event data ports are buffered. The buffer sizes and overflow policies can be configured per port using standardized AADL properties. Because of these similarities, we introduce the term *event-like* to refer collectively to event and event data ports. Inputs to data ports are not buffered; newly arriving data overwrites the previous value.

The periodic `tempSensor` thread measures the current temperature, e.g., from memory-mapped IO, which is not shown in the diagram, and transmits the reading on its `currentTemp` data port. If it detects that the temperature has changed since the last reading, then it sends a notification to the sporadic (that is, *event driven*) `tempControl` thread on its `tempChanged` event port. When the `tempControl` thread receives a `tempChanged` event, it will read the value on its `currentTemp` data port and compare it with the most recent set points. If the current temperature exceeds the high set point, it sends a command to the `fan` thread to turn `on`. Similarly, if the current temperature is low, it will send an `off` fan command. In either case, `fan` acknowledges whether it was able to fulfill the command by sending user-defined data types `FanAck.Ok` or `FanAck.Error` on its `fanAck` event data port.

AADL provides a textual view to accompany the graphical view. The following listing illustrates the *component type* declaration for the `tempControl` thread for the example above. The data transmitted over ports are of specific types, and properties such as `Dispatch_Protocol` and `Period` configure the tasking semantics of the thread.

```
thread TempControl
features
 currentTemp: in data port TempSensor::Temperature.i;
 tempChanged: in event port;
 fanAck: in event data port CoolingFan::FanAck;
 setPoint: in event data port SetPoint.i;
 fanCmd: out event data port CoolingFan::FanCmd;
properties
 Dispatch_Protocol => Sporadic;
 Period => 500 ms;  -- the min sep between incoming msgs
end TempControl;
```

The next listing illustrates integration of subcomponents. The body of process `TempControlProcess` type has no declared features because the component does not interact with its context in this simplified example. The implementation of `TempControlProcess` specifies subcomponents and subcomponent communications over declared connections between ports.

```
process TempControlProcess
  -- no features; no interaction with context
end TempControlProcess;

process implementation TempControlProcess.i
subcomponents
 tempSensor : thread TempSensor::TempSensor.i;
 fan : thread CoolingFan::Fan.i;
 tempControl: thread TempControl.i;
 operatorInterface: thread OperatorInterface.i;
connections
c1:port tempSensor.currentTemp -> tempControl.currentTemp;
c2:port tempSensor.tempChanged -> tempControl.tempChanged;
c3:port tempControl.fanCmd -> fan.fanCmd;
```

```
c4:port fan.fanAck -> tempControl.fanAck;
end TempControlProcess.i;
```

AADL provides many standard property sets including those used to configure the core semantics of AADL, e.g., thread dispatching or port-based communication. Developer-specified property sets enable one to define project-specific configuration parameters.

In this paper (following [19]) and in our AADL-HSM mechanization in Isabelle, we limit our semantics presentation to thread components since almost all of AADL's run-time services are associated with threads and port-based communication.[3]

3 Model Representation

The HAMR tool chain uses a similar approach to many AADL tools: it processes an AADL *instance model* [15, Section 4.1.6] as generated by the AADL OSATE Integrated Development Environment (IDE). The instance model indicates specific component implementations to be used in a system build, and it provides connection topologies directly in terms of thread component ports (thus "flattening" the model and emphasizing threads as the primary run-time entities of the system). Within its generated code, HAMR includes data structures representing model information such as thread interfaces, port declarations, connection information, and associated AADL properties referenced within the run-time system. AADL-HSM has a corresponding representation of model information centered around thread and port descriptors, *CompDescr* and *PortDescr* respectively,

record *CompDescr* =
  name :: *string*
  id :: *CompId*
  portIds :: *PortIds*
  dispatchProtocol :: *DispatchProtocol*
  dispatchTriggers :: *PortIds*
  compVars :: *Vars*

record *PortDescr* =
  name :: *string*
  id :: *PortId*
  compId :: *CompId*
  direction :: *PortDirection*
  kind :: *PortKind*
  size :: *nat*
  urgency :: *nat*

Like many modeling tools, HAMR autogenerates unique identifiers for model elements. Types are introduced for component identifiers and port identifiers in the code, and corresponding types *CompId* and *PortId* appear in the AADL-HSM definitions. *PortIds* is a type for a set of port identifiers. The top-level *Model* structure contains a mapping from component identifiers to component descriptors (similarly for ports),

record *Model* =
  modelCompDescrs :: (*CompId*, *CompDescr*) *map*
  modelPortDescrs :: (*PortId*, *PortDescr*) *map*
  modelConns :: *Conns*

Connections are represented as a map from a connection source *PortId* to a set of one or more target *PortIds*,

[3] Other categories of components, see [1], will be treated in later work.

type-synonym *Conns* = (*PortId*, *PortIds*) *map*

Functions *isInCDPID* and *isInCIDPID* are two of approximately 40 helper functions for accessing model elements in AADL-HSM specifications:

fun *isInCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
  where *isInCDPID m cd p* = (*p* ∈ *portIds cd* ∧ *isInPD*(*modelPortDescrs m* $ *p*))
fun *isInCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
  where *isInCIDPID m c* = *isInCDPID m* (*modelCompDescrs m* $ *c*)

These two examples are predicates that hold for a model m when a port with identifier p is an input port in a component with descriptor cd (or alternatively for a component with identifier c). The functions use Isabelle record, set, and map operations. Isabelle maps are defined as functions from a "*key*" domain to an option "*val*" domain of type "*key* ⇒ *option val*". We add an infix operator *$* whose application will reduce (via the Isabelle simplifier tactic) to a value v when a map lookup operation returns *Some v* (if lookup returns *None*, then the expression will fail to produce a canonical value).

A number of model well-formedness properties are included with the base model specification. The following example property specifies that connections only go from *out* ports to *in* ports of matching kinds:

definition *wf-Model-ConnsPortCategories* :: *Model* ⇒ *bool*
  where *wf-Model-ConnsPortCategories m* ≡
    (∀ *p* ∈ *dom* (*modelConns m*). *isOutPID m p* ∧
      (∀ *p′* ∈ (*modelConns m* $ *p*). *isInPID m p′* ∧ (*kindPID m p* = *kindPID m p′*)))

When HAMR translates a system representation into AADL-HSM, it also generates proofs that the model satisfies the well-formedness properties (which are first established by the HAMR OSATE IDE plug-in). Due to the architecture of the theory, these proofs only need to use the Isabelle simplifier tactics. This organization provides the foundation for specializations of the semantics to be defined (e.g., for AADL subsets that only use a particular set of features, or that require AADL properties relevant to a particular development pipeline). For example, by adding additional well-formedness properties, we can indicate the modeling sublanguage used on the DARPA CASE program for an seL4-based platform. [29] illustrates model-based specifications and well-formedness proofs for several system examples.

4 State Representation and Application Logic

A key contribution of [19] was a mathematical description of key elements of an AADL-based system's run-time state. In this section, we summarize our mechanization of those notions as well as various enhancements and extensions. Figure 2 presents the graphical summary from [19] of thread state concepts. Many of AADL's thread execution concepts are based on long-established task patterns and principles for achieving analyzeable real-time systems [9]. Following these principles, at each activation of a thread, the application code of the thread will abstractly compute a function from its input port values and local variables to

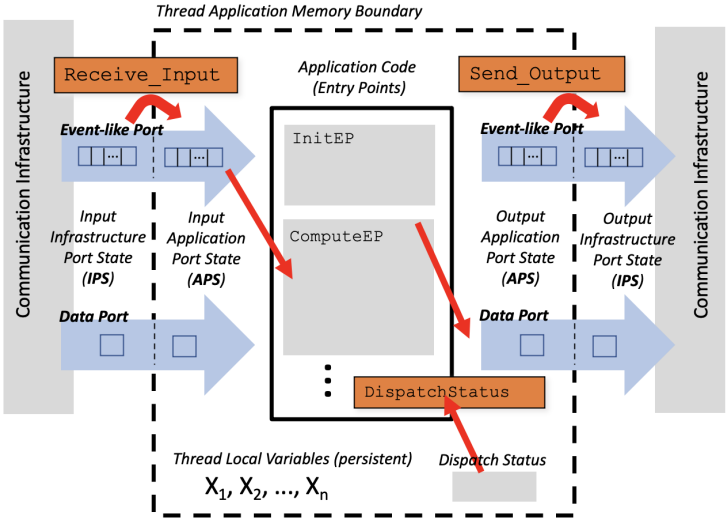

Fig. 2. Thread and Port State Concepts

output port values while possibly updating its local variables. In AADL terminology, *dispatching* a thread refers to the thread becoming ready for execution from a OS scheduler perspective, and the semantics for dispatch is determined by several thread properties specified in the AADL (captured formally as indicated in the definition *CompDescr* above). The thread `Dispatch_Protocol` property selects among several strategies for determining when a thread should be dispatched. In this paper, we consider only `Periodic`, which dispatches a thread when a certain time interval is passed, and `Sporadic`, which dispatches a thread upon arrival of messages to input ports specified as *dispatch triggers*. When a thread is dispatched, information describing the reason for its dispatch is stored in the thread's state and is retrievable via the `Dispatch_Status` RTS (diagrammed in Figure 2 and formalized below). For example, in a sporadic component, `Dispatch_Status` returns information indicating which port triggered the dispatch. This may be used in either the component application or infrastructure code to branch to a message handler method dedicated to processing messages arriving on the particular port.

Figure 2 illustrates that a thread's state includes the state of its ports, local variables, and dispatch status, and this is formalized in the *ThreadState* definition below.

record *'a ThreadState* =
  tvar :: *'a VarState*
  infi :: *'a PortState*
  appi :: *'a PortState*
  appo :: *'a PortState*
  info :: *'a PortState*
  disp :: *DispatchStatus*

datatype *DispatchStatus* =
  NotEnabled
  | *Periodic PortIds*
  | *Sporadic PortId* ∗ *PortIds*

In AADL-HSM, we represent thread local variable state *VarState* as a map from variable ids to values of an abstract type $'a$ as shown below.

type-synonym $'a$ *VarState* = (*Var*, $'a$) *map*

The polymorphic type $'a$ reflects the fact that the run-time semantics is orthogonal to the actual types of data being manipulated by application code. In an instantiation to the HAMR run-time, $'a$ would be instantiated to HAMR's universal type structure used to represent values constructed according to the AADL Data Modeling annex. This universal type structure enables HAMR to map values to and provide interoperability between the different programming language type systems supported by HAMR code generation.

One of the most important aspects of the thread state are the various buffers and memory blocks used to storing incoming and outgoing data for ports. In AADL-HSM, these are represented uniformally using values from the *PortState* type, which map from *PortId*s to $'a$ *Queue*s.

type-synonym $'a$ *PortState* = (*PortId*, $'a$ *Queue*) *map*

*Queue*s encapsulate lists with a bound and overflow policy as described in the AADL standard. We enhance the definitions in [19] with various properties proven about queue behavior.

Figure 2 illustrates that the state of each port is decomposed into the Infrastructure Port State (IPS) and the Application Port State (APS). The IPS represents the communication infrastructure's perspective of the port. The APS represents the thread application code's perspective of the port. The *ThreadState* definition includes input IPS and APS (*infi* and *appi*) and output IPS and APS (*info* and *appo*). The distinction between IPS and APS is used to represent AADL's notions of *port freezing* and *port variable* as presented in more detail in [14].

Typically, when a thread is dispatched, the component infrastructure uses the `Receive_Input` RTS (referenced in Section 5) to move one or more values from the IPS of input ports into the input APS. Then the component application code is called and the APS values remain "frozen" as the code executes. This provides the application a consistent view of inputs even though input IPS may be concurrently updated by communication infrastructure behind the scenes. The application code writes to the output APS throughout execution. Our intended design for this is that when the application code completes, the component infrastructure will call the `Send_Output` RTS to move output values from the output APS to the IPS, thus releasing the output values all at once to the communication infrastructure for propagation to consumers.

For input event data ports, the IPS typically would be a queue into which the middleware would insert arriving values following overflow policies specified for the port. For input data ports, the IPS typically would be a memory block large enough to hold a single value. For output ports, the IPS represents pending value(s) to be propagated by the communication infrastructure to connected consumer ports. At the component's external interface, this execution

pattern follows the *Read Inputs; Compute; Write Outputs* structure championed by various real-time system methods (e.g., [9]) enabling analyzeability.

The AADL standard indicates that a thread's application code is organized into entry points (e.g., subprograms that are invoked from the AADL run-time). For example, the *Initialize Entry Point* (`InitEP`) is called during the system's initialization phase, the *Compute Entry Point* (`ComputeEP`) is called during the system's "normal" compute phase. The behavior of entry point application logic is formalized in an *App* record as constraints on the thread state variables and the application port state.

record *'a App* =
  appInit :: *'a VarState* ⇒ *'a PortState* ⇒ *bool*
  appCompute :: *'a VarState* ⇒ *'a PortState* ⇒ *DispatchStatus* ⇒
    'a VarState ⇒ *'a PortState* ⇒ *bool*

The definitions indicate that a thread's `ComputeEP` behavior relates its input application port state, variable state, and dispatch status, to output application port state and possible updated variable state. The Initialize entry point has no "inputs"; it simply provides initial values for the output application port state and variable state. These definitions are strongly aligned with the GUMBO contract language for AADL. [21] illustrates how the Logika SMT-based verifier can automatically verify that a thread's component entry point code conforms to thread contracts. Alternatively, property-based testing can test that implementations conform to executable versions of the contracts [18]. Semantically, the contracts for each thread collectively give rise to contraints on the thread state as typed above. In a preliminary investigation, we have hand-translated GUMBO contracts into Isabelle using a shallow embedding. A fully automated translation of contracts will eventually enable us to carry out Isabelle proofs of system properties based on component behaviors derived from contracts that have been tested or proven, e.g., by Logika, to correctly summarize component behavior code.

Each thread is associated to its application logic behavior via a mapping from *CompId*s to *App*s.

type-synonym *'a CIDApp* = (*CompId*, *'a App*) *map*

As one example of a number of well-formed state properties, threads must only access their thread-local variable and port states as specified in the *Model*. This can be exspressed for a *CIDApp ca*, a *Model m*, a thread component *c* of the model *m* and an application *a* with *a* = *ca* $ *c*,

∀ *vs ps d ws qs*. *appCompute a vs ps d ws qs* ⟶
  (∀ *v*. *v* ∈ *dom vs* ∪ *dom ws* ⟶ *isVarOfCID m c v*)

We wish to distinguish the specifications that the developer supplies for an application (e.g., model information and thread application logic) from the state and execution logic of the AADL run-time and underlying execution platform. Accordingly, we introduce a record type *AppModel* for developer-supplied information that aggregates the structural model information (*Model*) and behavior specifications for each thread (*CIDApp*). Further separating the developer's structural specifications and behavior specifications permits AADL-HSM

to support an approach for model refinement where the structural model remains unchanged but the application logic can be considered at different abstraction levels.

record *'a AppModel* =
  appModel :: *Model*
  appModelApp :: *'a CIDApp*

Finally, the system state includes thread states (*systemThread*), the state of the communication substrate infrastructure (*systemComms*), and information for phasing and scheduling as discussed in Section 5.

record (*'u*, *'a*) *SystemState* =
  systemThread :: (*CompId*, *'a ThreadState*) *map*
  systemComms :: *'u*
  systemState :: (*CompId*, *ScheduleState*) *map*
  systemPhase :: *Phase*
  systemExec :: *Exec*

datatype *Phase* =
  Initializing | *Computing*
datatype *ScheduleState* =
  Waiting | *Ready* | *Running*
datatype *Exec* =
  Initialize CompId list |
  Compute CompId

systemComms is designed as an abstraction that can be instantiated to specific behavior rules based on the nature of a particular deployment platform (e.g., distributed communication via middleware, local communication via shared memory) and even different fault models regarding message loss, reordering, etc. This also supports the AADL standard's philosophy of not specifying precisely the implementation of communication, but instead constraining it (or stating assumptions about it), using various model properties. We only require particular instantiations to provide operations *comPush* and *comPull* to push data into the substrate and pull data out of the substrate. The two operations share the same signature. Taking the substrate state *'u*, a port state of a component *'a Portstate* and the port connections *Conns*, they yield a set of possible updates to the substrate and the port state. The signatures are symmetric and pass all available information to achieve a high degree of freedom for different instantiations.

record (*'u*, *'a*) *Communication* =
  comPush :: *'u* ⇒ *'a PortState* ⇒ *Conns* ⇒ (*'u* × *'a PortState*) *set*
  comPull :: *'u* ⇒ *'a PortState* ⇒ *Conns* ⇒ (*'u* × *'a PortState*) *set*

The least constraining substrate *CommonComm* permits any amount of pushing into and pulling out of the substrate (including doing nothing) that respect the queue capacities of the ports. The representation follows that of [22], storing tuples (*p, v, p', t*) where *p* is the sending port, *v* is a value, *p'* is the receiving port and *t* is a token that gives a unique identity to each message in the substrate. *CommonComm* does not impose any ordering of messages in the substrate.

definition *CommonComm* :: ((*PortId* ∗ *'a* ∗ *PortId* ∗ *nat*) *set*, *'a*) *Communication*
  where *CommonComm* = (| *comPush*= . . . , *comPull*= . . . |)

The motivation for this approach to expressing the substrate is to support verification of very basic properties using *CommonComm* and require other substrates to refine it. In particular, well-formedness properties of the state should

be inherited from *CommonComm* so that they do not have to be proved for other more specific substrates related to specific AADL execution and communication models (e.g., AADL's distinction between delayed and immediate communication).

5 System Behavior and Properties

As noted in [19], the current AADL standard underspecifies coordination between threads and the underlying scheduling and communications. The standard uses hybrid automata to specify constraints and timing aspects on the operational life-cycle of a thread (e.g., through initialization, compute, and finalization phases, along with mode changes and error recovery). Guarded transitions in the automata correspond to checks on the thread state, interactions with the scheduler, etc. Since the focus of the automaton is on a single thread, broader aspects of the system state, including the scheduling dimension and communication substrate, are not reflected in the standard.

In addition to specifying how HAMR interprets these concepts from the standard, AADL-HSM embodies a proposal to the broader AADL community for how these concepts may be formalized. Reflecting the standard's emphasis on the thread operational life-cycle, AADL-HSM first presents rules[4] corresponding to transitions in the standard's automata.[5] One important contribution of our work that supports reasoning about component and system behaviors is a formalization of application logic (which was not addressed in the standard) in the definition of a thread's execution steps.

Thread Behavior: The following rules illustrate some of the key aspects of thread execution. The AADL standard specifies that system execution is organized into (a) an *initialization* phase during which thread Initialize entry points are executed to initialize thread local variables and output ports and (b) a *compute* phase in which thread *Compute* entry points are executed according to the scheduling policy. We divide the thread execution step rules to match this phasing.

The main rule for thread execution in the initialization phase shown below "lifts" the thread application logic (which only constrains the thread local variables *tvar* and application's view of output ports *appo*) to the entire thread state.

inductive *stepInit* **for** $a :: 'a\ App$ **and** $t :: 'a\ ThreadState$
  where *initialize*: $appInit\ a\ vs\ ps \Longrightarrow stepInit\ a\ t\ (t(\!|\ tvar := vs,\ appo := ps\ |\!))$

The main rule for thread execution in the compute phase is shown below. The rule is parameterized on several auxiliary elements. t is the thread state for the thread undergoing a transition. Given the input infrastructure port state (*infi*)

[4] The rules are defined inductive to have access to the associated proof support that Isabelle offers. Sequences of transitions are modeled by transitive closure of the rules.

[5] Our life-cycle steps are simplified because we omit AADL mode switching and error recovery since HAMR does not currenly support these features.

for the thread, the function *cd* computes a set of dispatch status for the thread – indicating the possible scenarios in which it is dispatchable. We realize *cd* as a collection of Isabelle functions that formalizes the informally described rules in the AADL standard for thread dispatch. *ap* is the application logic for the thread. *ca* is a relation indicating that transitions between the thread's scheduling state (roughly corresponding to states in the life-cycle diagrams presented in the AADL standard). The three rules *dispatch*, *compute*, and *complete* are constrained to follow the life cycle evolution order. *dispatch* moves an enabled thread from its "waiting for dispatch" state to be ready for execution. The *receiveInput* auxillary rule formalizes the AADL RTS Receive Input as shown in Figure 2 used to move incoming data from the infrastructure ports of thread (*infi*) into the view of the application logic (in *appi*) The specific ports to receive data on are retrieved from the dispatch status via the *dispatchInputPorts* function (see [19] for a detailed discussion). The *compute* rule represents the execution step of thread task, and relationship between application input ports, variables, dispatch status, and output ports is determined by the application logic. In the post state, the input application port state is cleared. The *complete* rule releases the output port contents to the infrastructure, and moves the thread back to the *waiting* state. Together, the rules integrate the Receive Input RTS, the application logic, and Send Output RTS, to realize the Read Inputs; Compute; Write Outputs pattern described in Section 4.

inductive *stepThread* **for** *cd* :: *'a PortState* ⇒ *DispatchStatus set*
and *pk* :: *PortId* ⇒ *PortKind*
and *ap* :: *'a App*
and *ca* :: *ScheduleState* ∗ *ScheduleState*
and *t* :: *'a ThreadState* **where**

dispatch: ⟦ *ca* = (*Waiting*, *Ready*); *dsp* ∈ *cd* (*infi t*); *dsp* ≠ *NotEnabled*;
receiveInput pk (*dispatchInputPorts dsp*) (*infi t*) (*appi t*) *infi′ appi′* ⟧
⟹ *stepThread cd pk ap ca t* (*t*(| *infi*:= *infi′*, *appi*:= *appi′*, *disp*:= *dsp* |)) |

compute: ⟦ *ca* = (*Ready*, *Running*); *appCompute ap* (*tvar t*) (*appi t*) (*disp t*) *ws qs* ⟧
⟹ *stepThread cd pk ap ca t* (*t*(| *tvar*:= *ws*, *appo*:= *qs*,
appi:= *clearAll* (*dom* (*appi t*)) (*appi t*) |)) |

complete: ⟦ *ca* = (*Running*, *Waiting*); *sendOutput* (*appo t*) (*info t*) *appo′ info′* ⟧
⟹ *stepThread cd pk ap ca t* (*t*(| *appo*:= *appo′*, *info*:= *info′*,
disp:= *NotEnabled* |))

System Behavior: The following rules illustrate some of the key aspects of system execution. This presentation is a significant evolution in design beyond the non-mechanized rule excepts in [19]. The rules are parameterized on the combined model structure and application logic, the semantics for the communication substrate, and the system state. To connect with how AADL / HAMR was applied on the recent DARPA CASE project with Collins Aerospace, we show a variant of the rules instantiated to a static cyclic scheduling regime where *sc* is a data structure specifying the schedule.

inductive *stepSys* **for** *am* :: *'a AppModel*
and *cm* :: (*'u*, *'a*) *Communication*
and *sc* :: *SystemSchedule*

and $s :: ('u, 'a)$ *SystemState* **where**
initialize: ⟦ *isInitializing s*; *systemExec s* = *Initialize* (*c*#*cs*);
stepInit (*appModelApp am* $ *c*) (*systemThread s* $ *c*) *t* ⟧
⟹ *stepSys am cm sc s* (*s*(| *systemThread*:= (*systemThread s*)(*c*↦*t*),
systemExec:= *Initialize cs* |)) |
switch: ⟦ *isInitializing s*; *systemExec s* = *Initialize* []; *c* ∈ *scheduleFirst sc* ⟧
⟹ *stepSys am cm sc s* (*s*(| *systemPhase*:= *Computing*,
systemExec:= *Compute c* |)) |
push: ⟦ *isComputing s*; *systemThread s c* = *Some t*;
(*sb*, *it*) ∈ *comPush cm* (*systemComms s*) (*info t*) (*appModelConns am*) ⟧
⟹ *stepSys am cm sc s* (*s*(| *systemComms*:= *sb*,
systemThread:= (*systemThread s*)(*c*↦(*t*(| *info*:= *it* |))) |)) |
pull: ⟦ *isComputing s*; *systemThread s c* = *Some t*;
(*sb*, *it*) ∈ *comPull cm* (*systemComms s*) (*infi t*) (*appModelConns am*) ⟧
⟹ *stepSys am cm sc s* (*s*(| *systemComms*:= *sb*,
systemThread:= (*systemThread s*)(*c*↦(*t*(| *infi*:= *it* |))) |)) |
execute: ⟦ *isComputing s*; *systemExec s* = *Compute c*; *c*′ ∈ *scheduleComp sc* $ *c*;
stepThread (*computeDispatchStatus* (*appModel am*) *c*)
(*appModelPortKind am*) (*appModelApp am* $ *c*)
(*systemState s* $ *c*, *a*) (*systemThread s* $ *c*) *t* ⟧
⟹ *stepSys am cm sc s* (*s*(| *systemThread*:= (*systemThread s*)(*c*↦*t*),
systemState:= (*systemState s*)(*c*↦*a*),
systemExec:= *Compute c*′ |))

The *initialize* rule can apply when the system is in the initialization phase (*isInitializing s*): executing the Initialize Entry Point code of a thread component with id *c*, is represented by applying the application logic thread using the previously defined *stepInit* rule. Given that the Initialize entry point code does not read any state values, we are able to formally prove that the ordering of threads given by the thread id initialization list *Initialize cs* is irrelevant. The *switch* rule moves the system from initialization phase to the compute phase when the thread initialization list is empty.

Within the compute phase (*isComputing s*), in this most general (i.e., most abstract) version of the semantics, we do not constrain communication steps to a particular system schedule, which corresponds to the expected behavior, e.g., when using an independent middleware framework like the OMG Data Distribution Service (DDS) with its own notion of threading to move data between components. Thus, rules for moving data onto the communication substrate from a thread's output infrastructure ports (*push*) and for moving data off of the communication substrate to a thread's input infrastructure ports (*push*) are allowed to interleave arbitrarily with the execution steps of a thread (*execute*). In the *execute* rule, the thread with id *c* that appears next in the static schedule is stepped using the *stepThread* rule. The system state is updated to reflect the updated thread state, system state, and next thread component to be scheduled.

As indicated in Section 4, these rules are designed to be abstract, with various properties being provable about the most general case, but then refined to more specific notions of execution where stronger properties can be proved.

For example, the most general communication rules do not guarantee message delivery, or any notion of delivery fairness.

Given the rules above, notions of transitive system transitions and reachable states can be defined as follows.

definition *stepsSys* **where** *stepsSys am cm sc* = (*stepSys am cm sc*)**
definition *reachSys* **where**
reachSys am cm sc y ≡ ∃ *x*. *initSys* (*appModel am*) *x* ∧ *stepsSys am cm sc x y*

Property Framework: Following the semantics definitions above, properties of AADL models can be verified starting provided definitions of application logic (e.g., as might be automatically derived from AADL GUMBO contracts for each thread component). For instance, each application might establish some property P as described by *appInitProp*.

definition *appInitProp* **where**
appInitProp a P ≡ ∀ *s′ p′*. *appInit a s′ p′* ⟶ *P* (*s′*, *p′*)

This implies that a family of such properties $P\ c$ must be established for all thread components c on system level.

definition *sysInitProp* **where** *sysInitProp am P* ≡ ∀ *c* ∈ *appModelCIDs am*.
∀ *t t′*. *stepInit* (*appModelApp am* $ *c*) *t t′* ⟶ *P c* (*tvar t′*, *appo t′*)

A lemma of the accompanying theory characterises this relationship between application initialisations more precisely. Given a well-formed AADL instance model, a family of system initialisation properties is established, if we can prove that each initialisation establishes one of those. The latter is described as a family of verification conditions that must be established for the property on the system level to hold.

lemma *initSysFromApps*:
assumes *wf*: *wf-AppModel am*
and *vc*: ⋀*c*. *c* ∈ *appModelCIDs am* ⟹ *appInitProp* (*appModelApp am* $ *c*) (*P c*)
shows *sysInitProp am P*

This approach permits to reason as much as possible locally on the level of applications lifting these properties to the system level, ignoring its complexity at this stage. One can proceed similarly with the computation parts of applications, starting with application properties, and so on.

definition *appInvProp* **where** *appInvProp a I P* ≡
∀ *x x′ d p p′*. *I x* ∧ *appCompute a x p d x′ p′* ⟶ *I x′* ∧ *P* (*x′*, *p′*)

Supporting theories, e.g., on queues and states (i.e., maps to values or queues) are developed that are used to prove properties of the semantics. For instance, the proof that the order of the thread initialisations is irrelevant uses a theory on "merging" states that provides commutative merging operators on states. In this paper we focus on the presentation of the semantics and do not have space to discuss supporting theory.

6 Related Work

This work is part of a broader effort by members of the AADL community to increase the scope and precision of the modeling language's semantic definition. Most closely related is the work by Hugues et al. [22] that provides a mechanization of AADL model structures and standardized property (attribute) sets in the Coq proof assistant. A key goal of this ongoing work is to organize and document modeling language features to support the AADL standard committee and stakeholders, and the lay a foundation for eventually incorporating the formalism in the standard itself. The paper [22] focuses on syntactic and structural elements, but an accompanying open source repository also includes a number of interesting extensions and auxiliary material such as connections to Coq-specified real-time scheduling algorithms and initial work on representing the semantics of AADL threads in a Coq formalization of the DEVS discrete event simulator input language syntax. Our work is complementary in that it concentrates less on the structural aspects of AADL, but more on formalizing the execution state and run-time services (while clarifying aspects of these presented in the standard) to enable traceability and eventual proofs of correctness of AADL-driven code generation and run-time libraries. Other differences include our formalization of application logic and our initial framework for property-based reasoning (which are not present in [22]). This provides the foundation for future proofs of soundness of AADL contract languages such as GUMBO [21] and for mapping contracts down into code (e.g., as needed to fully justify property-based based testing against AADL model-based contracts as presented in our recent work [18]). Finally, we have placed more of an emphasis on designing aspects of the formalism as abstractions that can be formally refined towards specific platform and middleware semantics. There is no barrier to a deeper merging of the lines of work in either Coq or Isabelle.

While the above work has focused on mechanizations of AADL in theorem provers, there are many other contributions to the formal specification, analysis, and verification of AADL models and annexes. These works, whether implicitly or explictly, identify a target model checker or simulation framework with its own semantics, and then encode aspects of AADL into the semantics of the target framework. Some contributions focus on static semantics of models [4,31] while others consider run-time behavior and use model translation to extract executable specifications from AADL models, e.g., [6,7,10,16,33]. Rolland et al. [27] formalized aspects of AADL's coordination and timing behavior through the translation of AADL models into the Temporal Logic of Actions (TLA+) [25]. Many related works formalize a subset of AADL (e.g., for synchronous systems only) or focus on analyzing an aspect of the system, such as schedulability [30], behavioral [8,32], or dependability analyses [13].

7 Conclusion

The mechanized semantics that we have presented for AADL execution is substantial, and it provides a foundation for designing and verifying infrastructure

for AADL-aligned model-driven development. Its immediate impact is realized in the HAMR framework, which is being used in a number of industrial research projects. It complements other recent work [22] on mechanized AADL semantics by providing specifications and proofs for run-time state, including formalizations of key aspects of AADL run-time services, as well as property frameworks for component-level and end-to-end system reasoning. A valuable contribution is the holistic integration of these semantic aspects (component and system state, contracts, run-time services) to better support clean design and integration of accompanying tooling including contract-based verification of components [21], property-based testing of components [18], and multi-platform code generation [17]. Our approach emphasizes setting up abstract definitions, e.g., for communication, contracts, scheduling, that can subsequently be specialized via formal refinement to definitions for particular platforms and deployment scenarios.

AADL is a large modeling language, and we treat only a subset – chosing to omit at present more complicated features such as AADL modes and error recovery. In particular, we hope to include in the near future notions related to timing to be able to provide an interpretation for AADL's timing-related properties and to justify contract language features for time. We have found it challenging enough to obtain a formalization for our selected features and align them with our experience in code generation for multiple platforms and our implementation of contract-based verification and testing. We believe that our scope is reasonable because it aligns with features that have been used to support US defense-related research projects at Collins Aerospace that use AADL and HAMR to implement, e.g., prototypes of subsystems for the CH-47 Chinook military helicopter platform [11]. One aspect of our current formalism includes showing how the static cyclic scheduling used on those projects could be incorporated with other dimensions of the semantics.

8 Future Work

This work lays for the foundation for several important next steps.

Mechanized proofs of soundness for GUMBO contract framework: In previous work, Hatcliff et al. developed the AADL contract language that enables compositional reason about AADL systems. GUMBO is supported by both SMT-based verification [21] and property-based testing [18] tools that can be used to demonstate that component application code written in the Slang subset of Scala [26] conforms to GUMBO contracts. GUMBO is inspired by ideas from the AGREE [12] and BLESS [24] but puts a greater emphasis on aligning with the AADL run-time semantics and threading structure and with enabling application code to be verified against model-level component contracts. We are building the contract representations in AADL-HSM to formalize the semantics of GUMBO component contracts and to enhance the current notions of compositionality in GUMBO to support system-level properties and proofs. This includes both proving the soundness of the verification condition generation for

SMT-based reasoning [21] as well as the generation of executable contracts used in the accompanying property-based testing framework [18]. We are extending the HAMR generation of Isabelle AADL-HSM artifacts to include translation of GUMBO contracts. Although this would allow system properties of AADL models to be proved directly in Isabelle, we believe that greater end-user usability will be achieved if we use experience with the Isabelle-based definitions to design and prove the soundness of a highly automated deduction framework for application logic composition that would be incorporated as an extension to the Logika verification framework [26] (allowing developers to work directly in industrial IDEs instead of in Isabelle). AADL models and contracts from the Open PCA Pump medical device project [20] are being used as a driver for this work. It also seems possible to use AADL-HSM to address soundness of AADL-level contracts for secure information flow [2, 3].

Guidance for new platform backends: HAMR backends for seL4 and Linux are being refined and extended in ongoing industry projects. We are working on AADL-HSM extensions that specify the semantics of the AADL run-time at lower levels of abstraction corresponding to the different HAMR-based implementations on specific platforms. We intend that the AADL-HSM design will enable us to show these lower-level mechanization to be formal refinements of the general specifications. We are particularly interested over the long term in developing refinements to the Isabelle-based specifications of seL4. Our previous and ongoing collaborations with the seL4 team at ProofCraft and University of New South Wales on the DARPA CASE project are enabling preliminary planning for this effort [5].

Traceability and correctness of HAMR code generation: Many aspects of HAMR code generation are already aligned with our semantics, e.g., one can inspect the data types for HAMR's port state and thread state and the implementation of AADL RTS and observe the correspondence with AADL-HSM specifications. We are continuing to refactor both HAMR code generation to achieve stronger traceability, as well as the ability to translate arbitrary HAMR run-time states (as captured via logging) into Isabelle. Our ultimate and rather ambitious aim is to prove the soundness of HAMR code generation with respect to AADL-HSM. This would require utilizing, e.g., the C mechanized semantics infrastructured used in the seL4 proof base. HAMR code generation is factored through Slang [26] (a safety-critical subset of Scala support by the Logika verifier). Some aspects of the HAMR AADL run-time, e.g., the thread dispatch logic are written in a purely functional subset of Slang, while other sections (e.g., run-time service implementations) are amenable to Logika specification and verification. As a nearer-term goal, we are investigating translating purely functional Slang into Isabelle functions and making a stronger connection between Logika specifications and verification conditions and our Isabelle defintions.

References

1. Architecture analysis and design language (AADL), SAE AS5506 Rev. C (2017)

2. Amtoft, T., Dodds, J., Zhang, Z., Appel, A., Beringer, L., Hatcliff, J., Ou, X., Cousino, A.: A certificate infrastructure for machine-checked proofs of conditional information flow. In: Degano, P., Guttman, J.D. (eds.) Principles of Security and Trust. pp. 369–389. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
3. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby, Hoag, J., Greve, D.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 229–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
4. Backes, J., Cofer, D., Miller, S., Whalen, M.W.: Requirements analysis of a quad-redundant flight control system. In: NASA Formal Methods Symposium. pp. 82–96. Springer (2015)
5. Belt, J., Hatcliff, J., Robby, Shackleton, J., Carciofini, J., Carpenter, T., Mercer, E., Amundson, I., Babar, J., Cofer, D., Hardin, D., Hoech, K., Slind, K., Kuz, I., Mcleod, K.: Model-driven development for the seL4 microkernel using the HAMR framework. Journal of Systems Architecture (2022)
6. Berthomieu, B., Bodeveix, J.P., Chaudet, C., Dal Zilio, S., Filali, M., Vernadat, F.: Formal verification of AADL specifications in the Topcased environment. In: International Conference on Reliable Software Technologies. pp. 207–221. Springer (2009)
7. Berthomieu, B., Bodeveix, J.P., Dal Zilio, S., Dissaux, P., Filali, M., Gaufillet, P., Heim, S., Vernadat, F.: Formal verification of AADL models with fiacre and tina. In: ERTSS 2010-Embedded Real-Time Software and Systems. pp. 1–9 (2010)
8. Besnard, L., Gautier, T., Le Guernic, P., Guy, C., Talpin, J.P., Larson, B., Borde, E.: Formal semantics of behavior specifications in the architecture analysis and design language standard. In: Cyber-Physical System Design from an Architecture Analysis Viewpoint, pp. 53–79. Springer (2017)
9. Burns, A., Wellings, A.: Analysable Real-Time Systems: Programmed in Ada. CreateSpace (2016)
10. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP-application to the verification of real-time systems. In: International Conference on Model Driven Engineering Languages and Systems. pp. 5–19. Springer (2008)
11. Cofer, D.D., Amundson, I., Babar, J., Hardin, D.S., Slind, K., Alexander, P., Hatcliff, J., Robby, Klein, G., Lewis, C., Mercer, E., Shackleton, J.: Cyberassured systems engineering at scale. IEEE Security & Privacy **20**(3), 52–64 (2022)
12. Cofer, D.D., Gacek, A., Miller, S.P., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012). vol. 7226, pp. 126–140. Springer-Verlag, Berlin, Heidelberg (April 2012)
13. Feiler, P., Rugina, A.: Dependability modeling with the architecture analysis and design language (AADL). Tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering INST (2007)
14. Feiler, P.H.: Efficient embedded runtime systems through port communication optimization. In: 13th IEEE International Conference on Engineering of Complex Computer Systems. pp. 294–300. IEEE (2008)
15. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2013)
16. Hadad, A.S.A., Ma, C., Ahmed, A.A.O.: Formal verification of AADL models by event-b. IEEE Access **8**, 72814–72834 (2020)
17. Hatcliff, J., Belt, J., Robby, Carpenter, T.: HAMR: an AADL multi-platform code generation toolset. In: Leveraging Applications of Formal Methods, Verification

and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021. pp. 274–295. Lecture Notes in Computer Science, Springer (2021)
18. Hatcliff, J., Belt, J., Robby, Legg, J., Stewart, D., Carpenter, T.: Automated property-based testing fromaadl component contracts. In: Cimatti, A., Titolo, L. (eds.) Formal Methods for Industrial Critical Systems. pp. 131–150. Springer Nature Switzerland, Cham (2023)
19. Hatcliff, J., Hugues, J., Stewart, D., Wrage, L.: Formalization of the AADL runtime services. In: Leveraging Applications of Formal Methods, Verification and Validation - 11th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2022. pp. 105–134. Springer (2022)
20. Hatcliff, J., Larson, B.R., Carpenter, T., Jones, P.L., Zhang, Y., Jorgens, J.: The open PCA pump project: an exemplar open source medical device as a community resource. SIGBED Rev. **16**(2), 8–13 (2019)
21. Hatcliff, J., Stewart, D., Belt, J., Robby, Schwerdfeger, A.: An AADL contract language supporting integrated model- and code-level verification. In: Proceedings of the 2022 ACM Workshop on High Integrity Language Technology. HILT '22 (2022)
22. Hugues, J., Wrage, L., Hatcliff, J., Stewart, D.: Mechanization of a large DSML: an experiment with AADL and coq. In: 20th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2022. pp. 1–9. IEEE (2022)
23. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an os microkernel. ACM Transactions on Computer Systems **32**(1) (2014)
24. Larson, B., Chalin, P., Hatcliff, J.: BLESS: Formal specification and verification of behaviors for embedded systems with software. In: Proceedings of the 2013 NASA Formal Methods Conference. Lecture Notes in Computer Science, vol. 7871, pp. 276–290. Springer-Verlag, Berlin Heidelberg (2013)
25. Merz, S.: The specification language TLA+. In: Logics of specification languages, pp. 401–451. Springer (2008)
26. Robby, Hatcliff, J.: Slang: The sireum programming language. In: International Symposium on Leveraging Applications of Formal Methods. pp. 253–273. Springer (2021)
27. Rolland, J.F., Bodeveix, J.P., Chemouil, D., Filali, M., Thomas, D.: Towards a formal semantics for AADL execution model. In: Embedded Real Time Software and Systems (ERTS2008) (2008)
28. SAnToS Laboratory: HAMR project website. https://hamr.sireum.org (2022)
29. SAnToS Laboratory: AADL HAMR semantics mechanization – git repository. https://github.com/santoslab/AADL-HSM (2023)
30. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of aadl models. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. p. 8. IEEE (2006)
31. Stewart, D., Liu, J.J., Whalen, M., Cofer, D., Peterson, M.: Safety annex for architecture analysis design and analysis language. In: ERTS 2020: 10th European Conference Embedded Real Time Systems. p. 10 (2020)
32. Tan, Y., Zhao, Y., Ma, D., Zhang, X.: A comprehensive formalization of AADL with behavior annex. Scientific Programming **2022**, 2079880 (2022)
33. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: A verified model transformation. Journal of Systems and Software **93**, 42–68 (2014)