Formalization of the AADL Run-Time Services [⋆]

John Hatcliff[1], Jerome Hugues[2], Danielle Stewart[3], and Lutz Wrage[2]

[1] Kansas State University, Manhattan KS 66506, USA
[2] Software Engineering Institute
[3] Adventium Labs, Minneapolis, MN 55401, USA

Abstract. The Architecture and Analysis Definition Language (AADL) is an industry standard modeling language distinguished by its emphasis on strong semantics for modeling real-time embedded systems. These features have led to AADL being used in many formal-methods-oriented projects addressing critical systems. With regard to future directions in programming and systems engineering in general, questions naturally arise regarding how modeling language definitions should be documented so that the meaning of modeled systems can be made clear to all stakeholders. For example, the AADL standard describes Run-Time Services (RTS) that code generation frameworks can implement to realize AADL's standards-based semantics for thread dispatch and port-based communication. The documentation of these semantics in the AADL standard is semi-formal, allowing for divergent interpretations and thus contradictions when implementing analysis or code generation capabilities.
In this paper, we illustrate how key semantic elements of the AADL standard may be documented via a rule-based formalization of key aspects of the AADL RTS as well as additional services and support functions for realistic, interoperable, and assurable implementations. This contribution provides a basis for (a) a more rigorous semantic presentation in upcoming versions of the standard, (b) a common approach to assess compliance of AADL code generation and analysis tools, (c) a foundation for further formalization and mechanization of AADL's semantics, and (d) a more intuitive documentation of a system's AADL description via simulation and automatically generated execution scenarios.

1 Introduction

Model-based development tools are increasingly being used for system-level development of safety critical systems. The quality of model-based development depends greatly on the modeling language and its semantics. Models developed in languages with well-defined semantics can be a more faithful abstraction of

[⋆] This work is supported in part by Collins Aerospace, the U.S. Army Combat Capabilities Development Command, Aviation and Missile Center under Contract No.W911W6-20-C-2020, and the US Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. See the acknowledgements section for full details.

deployed systems, if the analysis correctly reflects the semantics and the system as implemented likewise reflects the semantics. Futhermore, having a rigorous semantics reduces the risk of misinterpretation when processing a model: analysis or code generation capabilities will deliver comparable results and can be composed with confidence.

AADL, the Architecture Analysis and Design Language [15], has historically been distinguished from other standardized modeling languages by having a semantics that precisely describe the architecture of safety-critical embedded systems. This definition for a focused semantics has motivated the use of AADL on multiple large scale industrial research projects, especially those that emphasize formal methods [11, 21, 30, 33, 38]. In AADL, component categories and other model elements have a comprehensive descriptions of intent and associated properties, which provides consistent definitions across models designed by different developers and supports early cross-vendor integration.

An important aspect of AADL's current standard definition is the inclusion of Run-Time Services (RTS). From an implementation view, the RTS are meant to be implemented as library functions in an AADL run-time that realize key actions of the infrastructure, including communication over ports and dispatching of threads. 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. From a semantics perspective, the RTS can be seen as basic operations in an abstract machine realizing the AADL run-time. In the current standard, RTS are presented in a highly descriptive narrative with suggestive notation for the type signature of the services, e.g., how they may be represented in code [23].

There are three main challenges with regard to RTS. First, there are no formal semantics for the services: the standard provides only an incomplete capture of run-time actions into standardized services. Second, the RTS are not sufficiently described to allow for a portable Application Programming Interface (API) to be derived. Third, AADL is highly configurable and the RTS are meant to support multiple computational models (e.g., synchronous, asynchronous, with configuration of details to support different application policies). Thus, existing formalisms cannot be used directly to support the general intent of the RTS. Previous work has formalized subsets of AADL [6] and other work has transformed AADL into formal languages for specific computational models [16, 37], but to our knowledge, a formalization of the general RTS semantics of AADL has never been defined.

We believe that appropriately documenting the semantics of the AADL RTS is an important step to meeting the challenges above. While one typically thinks of documentation as it applies to a specific model or program, in our setting, documentation is needed at the meta-level, i.e., for the *definition of the modeling language*. We argue in Section 3 that, with the future of programming being increasingly supported by modeling and meta-programming, arriving at effective solutions for documentation at different meta-levels of the programming framework is important for grounding the next generation of programming environments. Formally specifying the semantics of a modeling language is not a new idea. Rather, our contribution is to tackle the challenges of documenting key aspects of a modeling language that is being used for rigorous industry model-driven development in a manner that can support evolution of the modeling language standard and associated ecosystem. The main benefit of the work is to begin to fill a gap that, once closed, can provide a much more solid foundation for a community that is already invested in rigorous system engineering with AADL.

The contributions of this paper are as follows.

- Formalize the primary notions of thread state and communication state within an AADL system. These notions provide the foundation of conformity assessment by specifying what portions of a executing system's state must be documented for traceability to definitions within the AADL standard and observed for conformance testing based on execution traces.
- Develop rule-based formal documentation of the AADL standard's run-time services, providing a basis for soundness of AADL analysis and verification and the ability to develop conformity assessment for AADL run-time libraries.
- Identify additional run-time services and support functions needed to fill gaps in the current AADL standard.

Combined together, our goal is to provide an initial proposal for a formal documentation of services for AADL RTS for upcoming major revisions to the AADL standard. Expansion of this formalization can serve as a specification for establishing the correctness of AADL analyses and code generation. Our formalization aims to support the *general* nature of the AADL RTS and to provide a foundation for creating *specialized* semantic definitions for particular models of computation or platforms. From an implementation view, this provides a means by which implementers that optimize communication pathways or thread management can justify executions in their context to be sound refinements of the standardized general notions.

It is important that a proposal for aspects of a standardized AADL run-time be based on significant implementation experience with multiple code generation approaches, multiple languages, and multiple target platforms. The formalization presented here is informed by significant experience with the Ocarina and HAMR code generation frameworks. These frameworks have been used on industry and military research projects, target multiple real-time embedded platforms, and support multiple programming languages including C, Ada [3], Ada/Spark2014 [9], and Slang [19]. This work is supported by a technical report [18] that provides expanded treatment of semantics rules and examples.

We start with a background description of AADL in Section 2, This is followed by a summary of potential stakeholders of our proposed documentation and potential impact within the AADL ecosystem. Next, we summarize important concepts of run-time services in Section 4. Section 5 provides the static model definitions, and Sections 6, 7, 8 describe run-time services and semantics, and Section 9 illustrates these with examples. Section 10 presents related work, and Section 11 concludes.

2 AADL Background

SAE International standard AS5506C [23] 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.

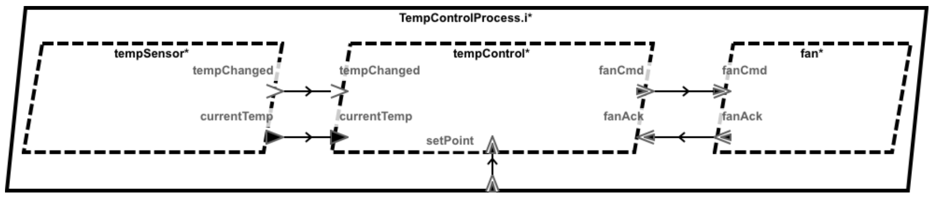

Fig. 1. Temperature Control Example (excerpts) – AADL Graphical View

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. The data passed 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). Inputs to event and event data ports are buffered. The buffer sizes and overflow policies can be configured per port using standardizes AADL properties. 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 (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 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 how architectural hierarchy is realized as an integration of subcomponents. The body of TempControlProcess type has no declared features because the component does not interact with its context in this simplified example. TempControlProcess implementation 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 properties, and allows definition of new properties. Examples of standard properties configure the core semantics of AADL, e.g., thread dispatching or port-based communication. User-specified property sets enable one to define project-specific configuration parameters.

In this paper, 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. Other categories will be treated in later work.

3 AADL Semantic Documentation Impact

In code-centric development, when there is a semantics-related question ("what's the behavior of this section of code?" or "what's the behavior of this language construct?"), the relevant code/construct can be executed and tested against expected results. While this approach may have its limitations (multiple compilers/interpreters may implement different semantics), it is generally effective in practice for the typical end user.

With a modeling language, the ability to understand semantics through experimentation seems less direct. Often modeling notations (at least the ones that we are considering for AADL) cannot be directly executed. Moreover, due to the intent to use modeling early in an incremental development process, models are often incomplete or partially developed. There is the possibility of understanding the semantics through generating code from the model or using a model simulator. However, for AADL and related modeling languages like SysML, there are currently no "reference implementations" of code generators or simulators. In addition, AADL is designed to be used with multiple languages, multiple RTOS, multiple middleware, etc. Therefore, a further challenge is to specify some portions of the semantic behavior as required or canonical, while at the same time admitting variations in behavior between platforms in a controlled way.

Accordingly, our long-term goal with this work is to provide a programming language-independent, RTOS-independent, middleware-independent formal documentation of semantic behavior. The formal description should then be refinable to describe lower-level variations of behavior and optimizations for different platforms. As refinements are developed, there must be a means to demonstrate conformity of implementations to refined semantic descriptions and of refined descriptions to the canonical (top-level) reference semantics. Since one of the objectives of AADL is to enable model analysis and verification, there is a need in the conformity assessment, tool qualification, and standardization contexts to precisely specify (a) soundness of verification and analysis tools, (b) the extent to which soundness of these tools is maintained in the presence of refinements to various platforms, (c) the specific situations (particular modeling features used, particular platform implementation strategies) in which tool soundness cannot be guaranteed. All of this complexity suggests that a mechanization of the semantic framework will be required to be able to appropriately manage the concerns and provide appropriate levels of assurance.

In this paper, our goals are to lay out some of the key concerns and provide an illustration of some of the most important aspects of the semantics. In ongoing work, we are pursuing a mechanization in Coq of the rules presented here, along with the broader coordination aspects of AADL. In some sense, a formal semantics lies at extreme end of the spectrum of documentation. Nevertheless, it seems crucial to enabling the next generation of model-based "programming" and system development.

In this section, we summarize the stakeholders for the semantic documentation presented in this paper, along with anticipated impact and connections within the AADL ecosystem.

3.1 Stakeholders

Modelers/System Engineers – the end users of the modeling language: End users often have questions about the intended execution semantics of AADL-based systems. For example, questions often arise about the semantics of port-based communication (e.g., exactly when are values available to read on input ports?, do unread values in input buffers carry over from one dispatch to another?, what is the ordering of values in an input buffer in the presence of multiple senders to the associated input port?, etc.) and threading (what should the structure of the application code be within a thread? when and how does thread application get executed?). These types of questions are addressed in the standard through informal textual descriptions that are distributed throughout different sections. Our aim is to develop more rigorous formal documentation that can be presented at one place in the standard, then referenced and assessed throughout. An example-based document that illustrates traces of threads, communication, and system composition in terms of the semantics rules (e.g., as presented in Section 9) may also be helpful.
Analysis/Verification Tool Providers: As mentioned above, one of the key objectives of AADL is to enable *analyzeable* models. There are many different forms of AADL analysis. Some of the most mature analysis areas include scheduling and timing analysis, information flow analysis, assume/guarantee contract specification and verification, and fault modeling and hazard analysis. To provide sound results, analysis and verification tools need to have a clear and unambigous specification of the aspects of the modeling language semantics. For example, information flow analysis needs to understand precisely how information flows between components and how information is propagated from component inputs to component outputs. For this area, even basic questions such as "is there an implicit acknowledgement of successful communication from a receiving component to a sending component" affects whether or not a potential "backflow" of information should be reported by the analysis. Current versions of AADL contract languages such as AGREE [12] and BLESS [26], make different assumptions about the behavior of component inputs outputs. Providing canonical documentation (as begun in Section 6) of the input/output behavior of components, structure of application code and infrastructure code, and the semantics of component composition is essential for reconciling the differences between these current verification frameworks and for establishing their soundness as needed for assurance in current industrial projects. Regarding hazard analysis in AADL's Error Modeling framework [22, 25], there is a need to clearly understand the nominal behavior of modeled systems to be able to clearly delineate and categorize faulty behavior.
Code Generation Tool Providers: Having clear, unambigous documentation of threading and communication behavior is essential for AADL code generation tools such as [7,17,27]. In fact, the original motivation of this work was to document the semantics of a set of APIs for AADL RTS that could be implemented in each of the code generation tools above. Work on verified compilers and code generators for modeling languages has demonstrated the feasibility of proving

the correctness of realistic code generation frameworks. To ensure that model-driven development strategies, which represent an emerging approach for next generation system development, can be inserted into tool chains for certified critical systems, a clear documentation of intended semantics that can also support conformity assessment is needed. A challenging aspect for AADL is that we must eventually support, not a single semantics, but rather a family of semantics that can be configured by AADL model `property` annotations.

3.2 AADL Ecosystem Synchronization

In addition to supporting the stakeholders above, behavioral documentation is needed to establish consistency across different aspects of the AADL standard itself. For example, AADL includes a Behavior Annex (BA) that provides a state-machine-based formalism for specifying thread behaviors. The semantics of BA needs to be aligned with the semantics of AADL entry points and port-based communication as presented in Section 6. AADL presents a variety of timing-related properties that are utilized by latency analysis and scheduling tools. Those timing properties need to be mapped onto documented behavioral semantics so as to eventually be reflected directly in abstract communication traces of the kind presented in Section 9. We have already discussed above the need to align AADL's Error Modeling framework with a documented semantics. In summary, work on the AADL standard itself as well as similar work in, e.g, the SysML community, can be made much more rigorous by exploring effective strategies for documenting system modeling language semantics.

4 Concepts

In this section, we enhance the presentation in the current AADL standard by describing the overall relationship between the AADL RTS, thread component code organization, and AADL communication infrastructure. Many of AADL's thread execution concepts are based on long-established task patterns and principles for achieving analyzeable real-time systems [8]. 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 output port values while possibly updating its local variables. In the general case, an AADL thread may explictly call the RTS to receive new inputs at any point in its execution. Yet, this would break the atomicity of a dispatch execution. We explictly forbid this in our formalization as this would introduce unsoundess in many AADL analyses and contract languages [12, 26]. Furthermore, this capability is barely used in practice. In AADL terminology, dispatching a thread refers to the thread becoming ready for execution from a OS scheduler perspective. 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

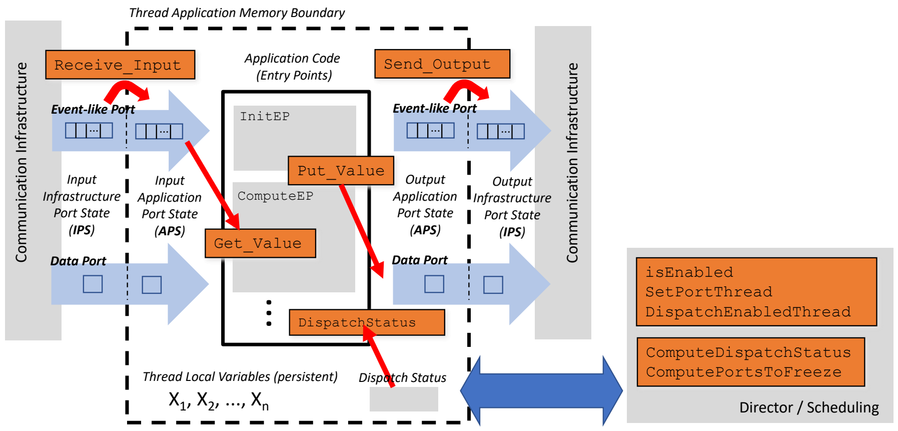

Fig. 2. Thread and Port State Concepts

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. 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. The state of each port is further 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 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 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 then 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. This release of output values is the key desired behavior. There are multiple possible implementations that achieve this behavior. 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., [8]) enabling analyzeability.

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.

The AADL standard specifies a collection of RTS including `Get_Value`, `Put_Value`, etc. that application code may use to access input and output APS.[4] We view these as less important semantically because they do not determine the overall semantic model of computation but rather suggest programming language-level idioms for accessing the APS.

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. Other entry points are defined for handling faults, performing mode changes, etc. We plan to address the higher-level coordination semantics for these phases in follow-on work. Multiple organizations of entry points are allowed. Here, we address a single `InitEP` and `ComputeEP` for each thread.

Overall, AADL emphasizes a specific canonical usage pattern of the RTS to achieve the Read; Compute; Write pattern described above. It also allows other usage patterns for flexibility and generality, but these may reduce the analyzeability of the system. For example, it allows the application code to call `Receive_Input` and `Send_Output` at any point and for multiple times. This "breaks" the view of the `ComputeEP` as a function from inputs to outputs. Our approach in this paper will be to define the semantics for key RTS (which hold regardless of useage pattern), and then provide a semantics for entry points that (a) adheres to the canonical usage pattern and (b) better supports formal reasoning.

5 Static Semantics

Prior to defining the semantic rules of AADL RTS, we introduce some key definitions and notations that map AADL concepts onto mathematical notations. The AADL standard includes informal rules that specify how model elements are organized in terms of containment hierarchy, allowed relationships (e.g., connections, bindings) with other components, and modeling patterns for system

[4] The AADL standard uses the term *port variable* to refer to the APS concept; "variable" suggests an application programming view of the port state. However, the port variable concept is somewhat ambigiously presented in the current standard and is intertwined with the binding to a particular programming language. For these reasons, we use a more mathematically oriented presentation of the concept in this paper and suggest that upcoming versions of the standard allow different programming language bindings to specify how they realize the concept in a particular language.

configuration and deployment. The static system model specifies components, ports, and connections. The intent of this section is not to exhaustively capture all static model information, but rather the information that influences the execution semantics presented in the following sections.

The root AADL system is instantiated and compiled into an abstract syntax tree. The resulting instance model, which we call $\mathcal{M}$, is the static AADL model consisting of an organized hierarchy of components. The set of all components of $\mathcal{M}$ we refer to as $\mathcal{C}$. The static instance model defines the hierarchical structure and interconnection topology of an operational system. An AADL instance model $\mathcal{M}$ represents the runtime architecture of an actual system that consists of application software components and execution platform components. A complete system instance that represents the containment hierarchy of the actual system is created by starting at the root component and instantiating all subcomponents recursively. We assume that $\mathcal{M}$ is completely instantiated and bound.

Next we define the structure of a generic AADL component. This definition covers both component type and implementations and derives from the implementation of the instance model from the OSATE AADL editor. Model information in $\mathcal{M}$ includes component descriptors that hold static information needed to determine the semantics of AADL components.

Definition 1. *A component $Comp \in \mathcal{C}$ is a tuple $(c_{id}, cat_C, \mathcal{F}, Prop, \mathcal{S})$ s.t.*
c_{id}: the unique component id,
cat_C: the component category,
$\mathcal{F}$: the set of features,
$Prop$: the set of properties associated with this component, and
S: the set of subcomponents.

Given that all component identifiers are unique at each layer of the instance model, navigation of the hierarchy is performed through top-down access of component identifiers starting from the root component r and traversing down the subcomponent hierarchy, e.g., $r.childId$.

An AADL component can have one of several categories cat_C, including software (e.g., `thread`, `process`), hardware (e.g., `processor`), and more. In this document, we focus on a subset of component categories: $cat_C \in \{thread, process, system\}$. Component properties $Prop$ specify relevant characteristics of the detailed design and implementation descriptions from an external perspective (e.g., a thread component's `Dispatch_Protocol`, as illustrated in Section 2). These can be viewed as rules of conformance between the described components and their implementation. A detailed summary of relevant component properties addressed in this document may be found in our related tech report [18].

A *feature* describes an interface of a component through which control and data may be provided to or required from other components.

Definition 2. *A feature $f \in \mathcal{F}$ is a tuple $(f_{id}, cat_F, d, type, Prop)$ such that:*
f_{id}: is the unique feature identifier,
cat_F: is the category of the feature,

d: is the direction of the feature, where $d \in \{in, out\}$*,*
type: is a component instance of the data type, and
$Prop$*: is the set of properties associated with this feature.*

The AADL standard defines several kinds of features. We focus on a subset of categories: $cat_F \in \{data, event, eventdata\}$[5]. Given that we focus on ports as a relevant subset of feature categories, we use the terms *port* and *feature* interchangeably. The direction d of a feature corresponds to either `in` or `out`. A feature *type* corresponds to a data component instance. Feature properties $Prop$ are configuration parameters that specify relevant characteristics of the feature and its implementation.

We refer to the global set of unique thread component identifiers as *ThreadIds*. We use the meta-variable $t \in \mathit{ThreadIds}$ to range over component identifiers. Port (feature) identifiers f_{id} can also be accessed globally in $\mathcal{M}$ similar to component identifiers. We use the meta-variable $p \in \mathit{PortIds}$ to range over port identifiers. To access the elements of the instance model $\mathcal{M}$, a number of helper methods and predicates are defined. For more detail on these predicates, see our companion tech report [18].

6 Threads

This section formalizes the primary elements of thread component state described in Section 4 along with associated AADL RTS. Section 7 addresses the aggregation of thread states into system states and RTS that are used in the coordination of thread actions with platform scheduling and communication.

6.1 Value Domains and Port Queues

Because the operational steps associated with RTS and entry points are agnostic to the values in port queues and user code variables, we assume a universal unspecified domain of values. To simplify the formalization, we adopt a uniform representation for IPS and APS for event, event data, and data ports: a queue q is a bounded sequence of values, $\langle v \rangle$ denotes a queue with a single data value v, e.g., as for a data port, and $\langle \cdot \rangle$ denotes an empty queue. $\langle q' \triangleright v \rangle$ represents a queue with v being the first item to be dequeued (head) and q' being the rest of the queue. This representation is specialized for data ports and event ports. For a data port, the queue size is always one and enqueueing overwrites the previous value. For event ports, queues hold 0 or more values, and $*$ denotes the presence of an event in a queue (e.g., $\langle * \rangle$ is a queue holding a single event). The maximum size of queues for a particular event and event data port is statically configurable via an AADL port property. To support AADL's specialized semantics for attempts to insert values in event and event data ports

[5] AADL other categories of features denote either abstract features or access to resources. They do not directly participate in the semantics of a thread and are omitted in this paper.

when a queue is full (configured by the overflow policy on a port), the function *enqueue*$(q, v,$ *overflow-policy*$)$ can be defined in a straightforward manner to return a new queue with v enqueued if the maximum size is not exceeded, else a value (e.g., the oldest in the queue) is dropped based on the given *overflow-policy* to make space for the newly enqueued value.

6.2 Thread State

Following the concepts of Figure 2, the state of each thread is formalized as a 6-element tuple

$$\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle.$$

I^i_t (IPS) maps each input port identifier of the component to a queue q holding buffered messages. For data ports, q is always non-empty and has a size of one. A^i_t (APS) maps each input port identifier of the component to a queue q representing the application code's view of frozen port values during its execution. AADL has configurable policies that enable a single value or a collection of values (up to the entire queue) to be dequeued from the IPS into APS. The representation of A^i_t as a mapping to queues is general enough to handle all of these variations. A^o_t (APS) maps each output port identifier for the component to a queue q of values representing the application code's view of output port. Typically, the application code will put 0 or 1 values onto the port during each dispatched execution. I^o_t (IPS) maps each input port identifier of the component to a queue q holding output values has been been moved from A^o_t, typically at the end of a entrypoint execution, to be available to the communication infrastructure for propagation to consumers. For any of these port state structures, the notation $A^i_t[p \mapsto q]$ denotes an updated port state structure that is like the original A^i_t except that port p p now maps to the queue q.

V_t represents a thread's local state, e.g., a mapping from variables to values, that may be accessed by the application code during entrypoint execution. Since manipulation of local variables is orthogonal to the semantics of the AADL RTS, we omit a more detailed formalization. D holds information, accessible during entry point execution, that provides information about the thread's current dispatch.

We assume that the thread state for a thread t is consistent with the static model specification of t (e.g., if t has an inport port p, there is an entry in the map I^i_t that provides a value/queue for p) and that consistency is maintained by the semantics rules. Such consistency properties are a straightforward formalization [18] and are omitted due to space constraints.

6.3 Dispatch Status RTS

Previous versions of the standard introduced a `Dispatch_Status` RTS, but left its behavior unspecified. The most recent version (a) introduces dispatch status as information held in the thread's state and (b) allows the status information to be platform independent but indicates that it should provide a basis to determine what triggered the thread's most recent dispatch. To move forward with formalizing these concepts, we introduce a basic set of status values for D to support periodic and sporadic dispatch protocols. $D \in \{\, \mathit{TimeTriggered}(ps), \mathit{EventTriggered}(p, ps), \mathit{Initializing}, \mathit{NotEnabled}\}$. $\mathit{TimeTriggered}(ps)$ indicates that a thread with a periodic dispatch protocol is enabled with ports ps to be frozen before application code executes (see the discussion in Section 4), $\mathit{EventTriggered}(p, ps)$ indicates that a thread with a sporadic dispatch protocol is enabled and that the triggering port is p with ports ps to be frozen, *Initializing* indicates that the thread is enabled in the *Initializing* system phase, and *NotEnabled* indicates that the thread is not enabled. Following the description in the most recent version of the standard, the `Dispatch_Status` can be formalized as a function that, when executed by a thread's application or infrastructure code returns the D information from the thread's current state. The returned D information can be used, e.g., by sporadic threads to select a message handler for the event-like port that triggered the dispatch.

6.4 Port RTS

The `Receive_Input` and `Send_Output` provide the interface to transfer messages between the IPS and APS. The following quote and accompanying interface description illustrates the level of descriptive detail for these services in the current AADL standard [1].

> A `Receive_Input` runtime service allows the source text of a thread to explicitly request port input on its incoming ports to be frozen and made accessible through the port variables. Any previous content of the port variable is overwritten, i.e., any previous queue content not processed by `Next_Value` calls is discarded. The `Receive_Input` service takes a parameter that specifies for which ports the input is frozen. Newly arriving data may be queued, but does not affect the input that thread has access to (see Section 9.1). `Receive_Input` is a non-blocking service.

Note that our introduction of I^i_t and A^i_t and IPS-APS terminology clarifies the distinction between the state of the infrastructure and the notion of port variable, respectively.

The interface for `Receive_Input` in version 2.3 of the standard is presented as follows:

```
subprogram Receive_Input
features
 InputPorts: in parameter <implementation-
     dependent port list>;
end Receive_Input;
```

The rules in the top portion of Figure 3 formalize the `Receive_Input` RTS. The rules are stated for a single port, but extend to a list of ports (matching the informal subprogram interface from the standard) in a straight-forward way.

The rules have the following relational structure:

$$\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D\rangle \xrightarrow{\text{RecInP}(p)}_t \langle {I^i_t}', {A^i_t}', A^o_t, I^o_t, V_t, D\rangle$$

ReceiveInputData :

$$\frac{\begin{array}{ll} \mathcal{M}.\mathit{isInPort}[t](p) & \\ \mathcal{M}.\mathit{isDataPort}[t](p) & \\ I^i_t(p) \;=\; \langle v \rangle & {A^i_t}' = A^i_t[p \;\mapsto\; \langle v \rangle] \end{array}}{\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle \overset{\text{RecInP}(p)}{\longrightarrow_t} \langle I^i_t, {A^i_t}', A^o_t, I^o_t, V_t, D \rangle}$$

ReceiveInputEventLikeOneItem :

$$\frac{\begin{array}{ll} \mathcal{M}.\mathit{isInPort}[t](p) & \\ \mathcal{M}.\mathit{isEventLikePort}[t](p) & \\ \mathcal{M}.\mathit{DequeuePolicy}[t](p) \;=\; \mathit{OneItem} & {I^i_t}' = I^i_t[p \;\mapsto\; q] \\ I^i_t(p) \;=\; \langle q \,\rhd\, v \rangle & {A^i_t}' = A^i_t[p \;\mapsto\; \langle v \rangle] \end{array}}{\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle \overset{\text{RecInP}(p)}{\longrightarrow_t} \langle {I^i_t}', {A^i_t}', A^o_t, I^o_t, V_t, D \rangle}$$

ReceiveInputEventLikeAllItems :

$$\frac{\begin{array}{ll} \mathcal{M}.\mathit{isInPort}[t](p) & \\ \mathcal{M}.\mathit{isEventLikePort}[t](p) & \\ \mathcal{M}.\mathit{DequeuePolicy}[t](p) \;=\; \mathit{AllItems} & {I^i_t}' = I^i_t[p \;\mapsto\; \langle\rangle] \\ \neg\, \mathit{isEmpty}(I^i_t(p)) & {A^i_t}' = A^i_t[p \;\mapsto\; I^i_t(p)] \end{array}}{\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle \overset{\text{RecInP}(p)}{\longrightarrow_t} \langle {I^i_t}', {A^i_t}', A^o_t, I^o_t, V_t, D \rangle}$$

ReceiveInputEventLikeEmpty :

$$\frac{\begin{array}{ll} \mathcal{M}.\mathit{isInPort}[t](p) & \\ \mathcal{M}.\mathit{isEventLikePort}[t](p) & \\ \mathit{isEmpty}(I^i_t(p)) & {A^i_t}' = A^i_t[p \;\mapsto\; \langle\rangle] \end{array}}{\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle \overset{\text{RecInP}(p)}{\longrightarrow_t} \langle I^i_t, {A^i_t}', A^o_t, I^o_t, V_t, D \rangle}$$

SendOutput :

$$\frac{\begin{array}{ll} & \mathit{newq} = \;\mathit{enqueue}(A^o_t(p), I^o_t(p), \\ & \qquad\qquad\qquad \mathcal{M}.\mathit{OverflowProtocol}[t](p)) \\ & {A^o_t}' = A^o_t[p \;\mapsto\; \langle\rangle] \\ \mathcal{M}.\mathit{isOutPort}[t](p) & {I^o_t}' = I^o_t[p \;\mapsto\; \mathit{newq}] \end{array}}{\langle I^i_t, A^i_t, A^o_t, I^o_t, V_t, D \rangle \overset{\text{SendOutP}(p)}{\longrightarrow_t} \langle I^i_t, A^i_t, {A^o_t}', {I^o_t}', V_t, D \rangle}$$

Fig. 3. *Receive Input* and *Send Output* RTS Rules (excerpts)

The $\xrightarrow{\text{RecInP}(p)}_t$ relational symbol indicates that thread t (typically component infrastructure code) is invoking the `Receive_Input` service on the port with identifier p. As a side-effect of the invocation, the state $\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle$ of thread t is transformed to a new state $\langle I_t^{i\prime}, A_t^{i\prime}, A_t^o, I_t^o, V_t, D\rangle$. The rule conclusions indicate that only the I_t^i and A_t^i portions of state are modified, e.g., by moving/copying values from the IPS into the APS. In source code, the call to the service would take the form RecInP(p) (here RecInP abbreviates the full name of the service). The other elements of the rule are implicit in the code execution context, e.g., t is the currently active thread with the associated thread states. Each of the clauses above the rule vertical bar represent antecedents to the rule (stacked, to conserve space). Clauses on the left side are typically preconditions or conditions selecting rule cases, whereas those on the right represent relationships between inputs and outputs. Each of the rules has an implicit side condition that its state elements I_t^i, etc. are compatible with the model's port declarations for t.

The rule for data ports (first rule) indicates that the single value held by a data port is copied from the infrastructure port state I_t^i to the application port state A_t^i. To conform with the AADL data port concept, the value is not removed from the infrastructure (I_t^i is not updated in the resulting state). The rule for event-like ports for a *OneItem* policy indicates that for a non-empty infrastructure port state I_t^i, a single item is dequeued and placed in the application port state. For the *AllItems* case, the entire contents of the infrastructure queue for p is moved to the application port state, leaving the infrastructure queue empty. Receive Input for p may be called when the infrastructure queue for p is empty, and this sets the application port to an empty queue.

At the bottom of Figure 3, the rule for `Send_Output` specifies that values are moved from the APS of the port p into the IPS, subject to developer specified overflow policy for p. There is only a single rule because the action is the same regardless the type of port.

6.5 Thread Entry Points

In Figure 4, there are two rules for each entry point category: the *AppCode* rules reflects the execution of "user's code", while the *Infrastructure* rules captures the code structure that AADL code generation tools would typically use for enforcing AADL's *Read; Compute; Write* emphasis. The distinct *AppCode* rules allow us to parameterize the semantics with the user code (thus, the use of the distinct $\Rightarrow$ symbol), i.e., our framework semantics for AADL RTS, entry point concepts, etc., is orthogonal to the application-specific semantics of the entry point user code.

The `InitEP` application code rule for a thread t formalizes the AADL standard's statements that (a) no input port values are to be read during the initialization phase (access to A_t^i is not provided to the code), (b) a thread's `InitEP` should set initial values for *all* of t's output data ports (i.e., after execution completes, all data port queues in $A_t^{o\prime}$ are required to have size of 1), and (c) sending initial values on event-like ports is optional. The `InitEP` infrastructure

InitEPAppCode :

$$\frac{\begin{array}{l}\forall p \in \mathcal{M}.\mathit{OutDataPorts}(t)\;.\;\mathit{size}(A_t^{o\prime}(p)) \;=\; 1\\ \forall p \in \mathcal{M}.\mathit{OutEventLikePorts}(t)\;.\;\mathit{size}(A_t^{o\prime}(p)) \;\le\; 1\end{array}}{(A_t^o,\, V_t) \Rightarrow_t^{\text{InitEP:app-code}} (A_t^{o\prime},\, V_t{}')}$$

InitEPInfrastructure :

$$\frac{\begin{array}{l}V_t^d \;=\; \mathit{DefaultVarValues}[t]\\ (A_t^o,\, V_t^d) \Rightarrow_t^{\text{InitEP:app-code}} (A_t^{o\prime},\, V_t{}')\\ ps \;=\; \mathcal{M}.\mathit{OutPorts}[t]\\ \langle I_t^i, A_t^i, A_t^{o\prime}, I_t^o, V_t', D\rangle \overset{\text{SendOutPL}(ps)}{\longrightarrow_t} \langle I_t^i, A_t^i, A_t^{o\prime\prime}, I_t^{o\prime}, V_t', D\rangle\end{array}}{\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \overset{\text{InitEP}}{\longrightarrow_t} \langle I_t^i, A_t^i, A_t^{o\prime\prime}, I_t^{o\prime}, V_t', D\rangle}$$

ComputeEPAppCode :

$$\frac{}{A_t^i,\, A_t^o,\, V_t,\, D \Rightarrow_t^{\text{ComputeEP:app-code}} A_t^{o\prime},\, V_t'}$$

ComputeEPInfrastructure :

$$\frac{\begin{array}{l}A_t^i,\, A_t^o,\, V_t,\, D \Rightarrow_t^{\text{ComputeEP:app-code}} A_t^{o\prime},\, V_t'\\ ps \;=\; \mathcal{M}.\mathit{OutPorts}[t]\\ \langle I_t^i, A_t^i, A_t^{o\prime}, I_t^o, V_t', D\rangle \overset{\text{SendOutPL}(ps)}{\longrightarrow_t} \langle I_t^i, A_t^i, A_t^{o\prime\prime}, I_t^{o\prime}, V_t', D\rangle\end{array}}{\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \overset{\text{ComputeEP}}{\longrightarrow_t} \langle I_t^i, A_t^i, A_t^{o\prime\prime}, I_t^{o\prime}, V_t', D\rangle}$$

Fig. 4. Thread Entry Point Semantics

rule indicates that when the scheduling framework invokes the entry point, the infrastructure code typically provides default values for all local variables, invokes the user code, then uses the `Send_Output` to release the thread's APS to the communication infrastructure for propagation. As the user code executes, it may set specific initial values for local variables (as indicated by the output $V_t{}'$).

The `ComputeEP` user code rule indicates that, abstractly, the user code computes a function *from* the input APS A_t^i, local variable state, and dispatch status *to* output APS values $A_t^{o\prime}$ and (updated) local variable state. A_t^o is present among the rule inputs even though it is *not* an input for the application code. The AADL standard indicates application code is prevented from reading the values of output ports, and A_t^o is only included in the rule inputs to enable the Director to provide initial values for output ports in the *SetPortState* rule of Figure 6 according to configurable policies. The `ComputeEP` infrastructure rule indicates that when the scheduling framework invokes the entry point, the infrastructure code will typically first invoke the user code, then use the `Send_Output` to move values from thread's output APS to IPS (receiving values for input ports is addressed in the dispatching rules of Section 7).

Overall, these rules enhance the AADL standard by clarifying the specific portions of thread state that entry points may read or write and indicating other semantic properties. The infrastructure rules illustrate how future versions of the standard may present code pattern options for enforcing important semantic properties of AADL.

7 Director

To support model analyzability, as well as a variety of embedded execution platforms, our formulation includes structured real-time tasks, without specifying concrete scheduling and communication implementations. This supplements the current AADL standard which underspecifies the necessary coordination between the thread 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 practice, parts of the automata semantics would be realized by a "director" that coordinates multiple types of code – including RTS used in thread infrastructure code, thread application code, and underlying platform scheduling and communication to achieve the semantics of the modeled system. A standardized description of this system-wide director concept would facilitate both interoperability and analyzability.

One of the contributions of our formalization is to fill gaps in the standard with initial definitions of the notion of the system state, and provide rules that characterize how the actions of individual threads, AADL run-time, scheduling, and communication evolve the system state (see [18] for a fuller treatment). Due to space constraints, we focus the presentation here on aspects related to the goals for state and rules above and omit timing aspects.

7.1 System State

For the subset of AADL that we are addressing, the state of an AADL system is a 4-tuple

$$\langle Phs, Thrs, Schs, Comms \rangle.$$

The *system phase* $Phs \in \{Initializing, Computing\}$ indicates the current system phase (initialization entry points are being executed, or compute entry points are being executed). *Thrs* maps each thread identifier t to the thread's state (as defined in Section 6). *Schs* maps each thread identifier t to the scheduling state of the thread (*WaitingForDispatch*, *Ready*, or *Running*). *Comms* represents the state of the communication infrastructure.

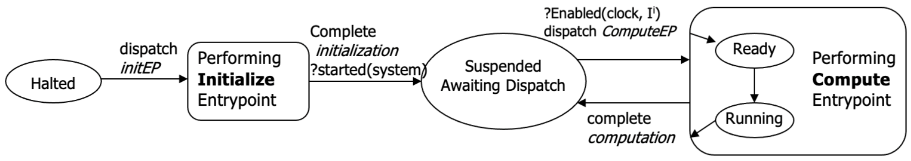
Halted
dispatch *initEP*
Performing **Initialize** Entrypoint
Complete *initialization* ?started(system)
Suspended Awaiting Dispatch
?Enabled(clock, I^i) dispatch *ComputeEP*
Ready
Running
Performing **Compute** Entrypoint
complete *computation*

Fig. 5. Thread Operational Life-Cycle

$$\mathit{SetPortState}:\quad \frac{\begin{array}{l} A_t^{i\prime} \;=\; \mathit{InitInAppPorts}(t) \\ A_t^{o\prime} \;=\; \mathit{InitOutAppPorts}(t) \\ ps \;=\; \mathit{PortsToFreeze}(t, D) \\ \langle I_t^i, A_t^{i\prime}, A_t^{o\prime}, I_t^o, V_t, D\rangle \xrightarrow{\mathrm{RecInPL}(ps)}_{t} \langle I_t^{i\prime}, A_t^{i\prime\prime}, A_t^{o\prime}, I_t^o, V_t, D\rangle \end{array}}{\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \xrightarrow{\mathrm{SetPortState(t)}}_{\mathrm{Dir}} \langle I_t^{i\prime}, A_t^{i\prime\prime}, A_t^{o\prime}, I_t^o, V_t, D\rangle}$$

$$\mathit{Dispatch\ Enabled}:\quad \frac{\begin{array}{l} \mathit{Phs}(t) \;=\; \mathit{Computing} \\ \mathit{Schs}(t) \;=\; \mathit{WaitingForDispatch} \\ \mathit{Thrs}(t) \;=\; \langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \\ \mathit{ComputeDispatchStatus}(\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle) \;=\; D' \\ D' \neq \mathit{NotEnabled} \\ \langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D'\rangle \xrightarrow{\mathrm{SetPortState(t)}}_{\mathrm{Dir}} \langle I_t^{i\prime}, A_t^{i\prime}, A_t^{o\prime}, I_t^o, V_t, D'\rangle \\ \mathit{Thrs}' \;=\; \mathit{Thrs}[t \mapsto \langle I_t^{i\prime}, A_t^{i\prime}, A_t^{o\prime}, I_t^o, V_t, D'\rangle] \\ \mathit{Schs}' \;=\; \mathit{Schs}[t \mapsto \mathit{Ready}] \end{array}}{\langle \mathit{Phs}, \mathit{Thrs}, \mathit{Schs}, \mathit{Comms}\rangle \xrightarrow{\mathrm{DispatchEnabledThr(t)}}_{\mathrm{Dir}} \langle \mathit{Phs}, \mathit{Thrs}', \mathit{Schs}', \mathit{Comms}\rangle}$$

Fig. 6. Director Services and System Actions (excerpts)

Figure 5 shows an adaption of the automata from Figure 5 (Section 5.4.2) of the AADL standard [1] that is simplified to our setting by removing notions related to mode switches, error recovery, and preemption. As discussed above, we aim to clarify the intent of such figures in the standard by providing formal definitions of the transitions, introducing new standardized services and revising the automata concepts when necessary. For example, consider the dispatch transition from the thread Suspended Awaiting Dispatch state to the Performing Compute Entrypoint state. The standard's narrative indicates that the dispatch action causes the thread's input ports to be frozen and that the thread is released to the scheduling framework for the compute entry point to be scheduled and executed. The execution of the dispatch action is conditioned on the notion of *enabled*, which itself is based on a non-trivial set of conditions concerning pending messages in port queues (sporadic components) and timing constraints (periodic components).

Figure 6 proposes a formalization of a new RTS *DispatchEnabledThread* to be used by the AADL Director. This rule is enabled when the system is in the *Computing* phase and the thread's scheduling state is *WaitingForDispatch*. The appropriate thread state is retreived from the system threads, and a new proposed auxiliary service `Compute_Dispatch_Status` is used to compute this dispatch status of the thread. If the thread is enabled, its input queue contents are transferred from the IPS to APS. The thread's scheduling state is set to *Ready* to indicate to the underlying scheduling framework that the thread is available for scheduling.

8 Communication

The design philosophy of the AADL standard is to integrate with existing communication frameworks including local communication using shared memory,

$$\begin{array}{l}
\mathit{CommOutput}: \\
\dfrac{\begin{array}{l}
\mathcal{M}.\mathit{isOutPort}[t](p) \\
\mathit{Thrs}(t) = \langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \\
I_t^o(p) = \langle q \rhd v\rangle \\
\mathcal{M}.\mathit{ConnDests}[t](p) = \mathit{ps} \\
\mathit{makeMessages}(p, \mathit{ps}, v) = M \quad I_t^{o\prime} = I_t^o[p \mapsto q] \\
\mathit{Thrs}' = \mathit{Thrs}[t \mapsto \langle I_t^i, A_t^i, A_t^o, I_t^{o\prime}, V_t, D\rangle] \\
\mathit{Comms}' = \mathit{Comms} \cup M
\end{array}}{\langle \mathit{Phs}, \mathit{Thrs}, \mathit{Schs}, \mathit{Comms}\rangle \xrightarrow{\text{CommOutCP}(t,p)} \langle \mathit{Phs}, \mathit{Thrs}', \mathit{Schs}, \mathit{Comms}'\rangle} \\
\mathit{CommInput}: \\
\dfrac{\begin{array}{l}
(p_s, v, p) \in \mathit{Comms} \\
\mathit{Thrs}(t) = \langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle \\
\mathit{newq} = \mathit{enqueue}(v, I_t^i(p), \mathcal{M}.\mathit{OverflowProtocol}[t](p)) \\
I_t^{i\prime} = I_t^i[p \mapsto \mathit{newq}] \\
\mathit{Thrs}' = \mathit{Thrs}[t \mapsto \langle I_t^{i\prime}, A_t^i, A_t^o, I_t^o, V_t, D\rangle] \\
\mathit{Comms}' = \mathit{Comms} - (p_s, v, p)
\end{array}}{\langle \mathit{Phs}, \mathit{Thrs}, \mathit{Schs}, \mathit{Comms}\rangle \xrightarrow{\text{CommInCP}(t,p)} \langle \mathit{Phs}, \mathit{Thrs}', \mathit{Schs}, \mathit{Comms}'\rangle}
\end{array}$$

Fig. 7. Communication Substrate Rules

middleware for distributed message passing, and scheduled communication such as ARINC 653. The standard provides no API or semantic representation for communication actions – it only provides properties that are used to state constraints/assumptions about communication frameworks. To move toward a formalization that supports this philosophy, we introduce a small number of very general rules whose actions and orderings within system traces that can subsequently be formally contrained to reflect different communication implementations. In particular, the rules are general enough to handle through subsequent refinements both distributed communication as well as local communication where queue state representations can be optimized, e.g., using the same storage for the output IPS of sender components with the input IPS of receiving components as has been done for AADL code generation for microkernels [17]. Space constraints do not allow a full development of this concept here. Instead we expose only enough concepts to support illustration of the port RTS actions in Section 9.

Figure 7 presents rules reflecting actions carried out by the communication substrate (middleware). The first rule reflects the substrate's transfer of messages M from a sending thread onto the communication substrate *Comms*. The second rule reflects the substrate's delivery of a message (p_s, v, p) from a sender port p_s into the input infrastructure port state of a receiver port p.

For the first rule, when the sender port p has an output infrastructure port queue $\langle q \rhd v\rangle$ with value v at its head, the substrate will form a collection of messages M, one message for each destination port in $\mathcal{M}.\mathit{ConnDests}[t](p) = \mathit{ps}$ as determined by static model connections. A new version of the output instructure port state with v dequeued is formed, and the newly created messages M are placed on the communication substrate.

With these general concepts, contraints applied to system traces that include these rule instances can accomodate concepts like immediate or delayed communication between input/output, order preserve delivery versus possible reorderings, and lossy vs non-lossy communication.

9 Example Traces

The section illustrates the AADL RTS rules using a simple execution scenario based on the example of Section 2. Steps in both the *Initializing* and *Computing* phases are shown for the `TempSensor` and `TempControl` components. These examples focus on the manipulation of thread states. See [18] for additional examples that include a broader treatment of system-level execution and communication. To validate that our rules reflect a realistic implementation of an AADL run-time, the HAMR AADL code generation framework [17] has been enhanced to emit trace states and traceability information that aligns with the rules in the previous sections. That feature generates outputs that are equivalent to the type set trace information presented in this section.

TempSensor Initialize Entry Point: Consider first the *InitEPAppCode* rule from Figure 4 capturing the execution of the application code for the Initialize Entry Point as applied to the TempSensor component. Given the values below for the initial output application port state, which includes empty queues for both output ports and default variable values supplied from the *InitEPInfrastructure* rule context (there are no thread local variables for this component),

$$\begin{aligned} A^o_t &= [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle] \\ V_t &= [] \end{aligned}$$

executing the application code for Initialize Entry Point is represented by the following relation, reflected in the *InitEPAppCode* rule.

$$(A^o_t,\, V_t) \Rightarrow^{\text{InitEP:app-code}}_t ([\texttt{currentTemp} \mapsto \langle 73.0\rangle, \texttt{tempChanged} \mapsto \langle\rangle],\, [])$$

This reflects the fact that the application code initializes the `currentTemp` port with the value 73.0 via the `Put_Value` run-time service, but does not put an event on the `tempChanged` port (this is not done until the Compute Entry Point where temperature values are retrieved from the sensor hardware). Making the assumption that the fan inherently knows the initial safe state is acceptable in this nominal fault-free safe state startup scenario, but would need to be discharged in other safety-critical design scenarios.

In the context of the Figure 4 *InitEPInfrastructure* rule, which formalizes the component *infrastructure's* invocation of the application code for the Initialize Entry Point, when the above structures are placed in the context of the `TempSensor` thread state, we have the following state thread elements, representing the state of the `TempSensor` thread after the execution of the *application code* (the tuple element values on the right side of the vertical bar reflect the instantiation of the meta-variables on the left side that are used in the

InitEPInfrastructure rule).

$$\begin{array}{r|l}
I_t^i & \langle [], \\
A_t^i & [], \\
A_t^{o\prime} & [\mathtt{currentTemp} \mapsto \langle 73.0 \rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
I_t^o & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
V_t' & [], \\
D & \mathit{Initializing} \rangle
\end{array}$$

Then, the invocation of the SendOutput RTS moves the `currentTemp` value to the output infrastructure port state to obtain the following thread state at the completion of the execution of the *infrastructure* for the Initialize Entry Point.

$$\begin{array}{r|l}
I_t^i & \langle [], \\
A_t^i & [], \\
A_t^{o\prime\prime} & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
I_t^{o\prime} & [\mathtt{currentTemp} \mapsto \langle 73.0 \rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
V_t' & [], \\
D & \mathit{Initializing} \rangle
\end{array}$$

The presence of value 73.0 in the currentTemp port of the $I_t^{o\prime}$ output infrastructure port state represents the fact that it has been made available to the communication substrate for propagation to consumers. .

TempControl Initialize Entry Point: Following a similar sequence of rules for the `TempControl` Initialize Entry Point yields the following thread state.

$$\begin{array}{r|l}
I_t^i & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle, \\
 & \mathtt{setPoint} \mapsto \langle\rangle, \mathtt{fanAck} \mapsto \langle\rangle] \\
A_t^i & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle, \\
 & \mathtt{setPoint} \mapsto \langle\rangle, \mathtt{fanAck} \mapsto \langle\rangle] \\
A_t^{o\prime\prime} & [\mathtt{fanCmd} \mapsto \langle\rangle] \\
I_t^{o\prime} & [\mathtt{fanCmd} \mapsto \langle\rangle] \\
V_t' & [\ \mathtt{latestTemp} \mapsto 73.0, \mathtt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \ \mathtt{currentFanState} \mapsto \mathtt{Off} \\
D & \mathit{Initializing}
\end{array}$$

The primary difference is that the `TempControl` thread initializes thread local variables (e.g., set points are iniitialized to low and high values $(70.0, 75.0)$), but does not put any initial values on its output ports (i.e., the queue for `fanCmd` is empty).

Before the system transitions from the *Initializing* to the *Computing* phase, the initial value placed on the `TempSensor` `currentTemp` output port is propagated to connected input port of `TempControl` according to the rules of Figure 7.

Completed Initialization: Following these communication steps, the system transitions to the *Computing* phase, with the following states for the TempSensor

$$\begin{array}{r|l}
I_t^i & \langle [], \\
A_t^i & [], \\
A_t^o & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
I_t^o & [\mathtt{currentTemp} \mapsto \langle\rangle, \mathtt{tempChanged} \mapsto \langle\rangle], \\
V_t & [], \\
D & \mathit{NotEnabled} \rangle
\end{array}$$

and TempControl thread components.

$$\begin{array}{r|l}
I_t^i & [\texttt{currentTemp} \mapsto \langle 73.0\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^i & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\
I_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\
V_t & [\,\texttt{latestTemp} \mapsto 73.0, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \;\texttt{currentFanState} \mapsto \texttt{Off} \\
D & \mathit{NotEnabled}
\end{array}$$

The `TempSensor` `currentTemp` out data port has been initialized and its value propagated to the infrastructure port state I_t^i for the `TempControl` `currentTemp` input data port. The thread local variables of `TempControl` have been initialized. Director rules (omitted due to space constraints) set the dispatch status of each thread to *NotEnabled*.

TempSensor Compute Entry Point: The `TempSensor` compute entry point is dispatched periodically as discussed in Section 2. Its entry point application code reads the current temperature from a hardware sensor (how this is modeled is not discussed in this paper), places the value on the `currentTemp` out data port and a notification event on `tempChanged` out event port. Assume the physical sensor responds with a temperature value of 78.3.

We use the `TempControl` component to illustrate the details of compute entry point rules, and simply note that at the end of a similar application of the rules for `TempSensor`, its thread state is the following.

$$\begin{array}{r|l}
I_t^i & \langle [], \\
A_t^i & [], \\
A_t^{o\prime\prime} & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle], \\
I_t^{o\prime} & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle *\rangle], \\
V_t' & [], \\
D & \mathit{NotEnabled}\rangle
\end{array}$$

Here we see that an event is queued in the `tempChanged` infrastructure port and the value 78.3 is held in the `currentTemp` infrastructure data port.

Communication: If we are following AADL's "immediate" port communication property, at this point that communication infrastructure would propagate values from the `TempSensor` output ports to the `TempControl` input ports. This results in the following `TempSensor` thread state below in which `currentTemp` and `tempChanged` have been communicated and no longer appear in I_t^o.

$$\begin{array}{r|l}
I_t^i & \langle [], \\
A_t^i & [], \\
A_t^o & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle\rangle], \\
I_t^o & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle], \\
V_t & [], \\
D & \mathit{NotEnabled}\rangle
\end{array}$$

At the other end of the port connections, the values have been placed in the input infrastructure ports I_t^i of the `TempControl` thread (the previous value of 73.0 in the infrastructure `currentTemp` data port has been overwritten by the value of 78.3).

$$\begin{array}{r|l} I_t^i & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle * \rangle, \\ & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\ A_t^i & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\ & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\ A_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\ I_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\ V_t & [\,\texttt{latestTemp} \mapsto 73.0, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\ & \ \texttt{currentFanState} \mapsto \texttt{Off} \\ D & \mathit{NotEnabled} \end{array}$$

TempControl Dispatch and Compute Entry Point: `TempControl` is a sporadic thread. In the current execution, its dispatch is triggered by the arrival of an event on the `tempChanged` port. Dispatching is formalized by the rules in Figure 6.

Conditions in premises of the *Dispatch Enabled* rule ensure that the system is in the *Computing* phase and that the `TempControl` scheduling state is *WaitingForDispatch*. Then an auxiliary function is called to determine the dispatch status of the thread.

$$\begin{array}{l} \mathit{ComputeDispatchStatus}(\langle I_t^i, A_t^i, A_t^o, I_t^o, V_t, D\rangle) = \\ \quad \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\}) \end{array}$$

This indicates that thread dispatch has been triggered by the arrival of the `tempChanged` event and that, following the AADL standard's rules for port freezing, the `tempChanged` is to be frozen since it is the triggering event port and `currentTemp` is to be frozen because it is a data port (data ports are frozen by default).

Since the computed dispatch status indicates that the thread is enabled, the *SetPortState* rule prepares the port states for entry point execution. First, auxiliary functions are used to initialize the input and output application port state as follows (where $t = \texttt{TempControl}$) :

$$\begin{array}{r|l} A_t^{i\prime} & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \ = \mathit{InitInAppPorts}(t) \\ & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\ A_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle\rangle] = \mathit{InitOutAppPorts}(t) \end{array}$$

An auxiliary function is also used to extract the set of ports to freeze.

$$\begin{array}{r|l} ps & \{\texttt{tempChanged}, \texttt{currentTemp}\} = \\ & \quad \mathit{PortsToFreeze}(t,\ \mathit{EventTriggered}(\texttt{tempChanged}, \\ & \qquad\qquad\qquad\qquad\qquad\qquad \{\texttt{tempChanged}, \texttt{currentTemp}\})) \end{array}$$

Given the thread state at this point, which instantiates the metavariables in the *SetPortState* rule as follows,

$$\begin{array}{r|l} I_t^i & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle * \rangle, \\ & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\ A_t^{i\prime} & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\ & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\ A_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle\rangle] \\ I_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\ V_t & [\,\texttt{latestTemp} \mapsto 73.0, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\ & \ \texttt{currentFanState} \mapsto \texttt{Off} \\ D & \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\}) \end{array}$$

and given the list of ports to freeze *ps* as instantiated above, applying the rule for the Receive Input RTS yields the following updated thread state (instantiating the metavariables in the *SetPortState* rule)

$$\begin{array}{r|l}
I_t^{i\prime} & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^{i\prime\prime} & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle * \rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle\rangle] \\
I_t^{o} & [\texttt{fanCmd} \mapsto \langle\rangle] \\
V_t & [\,\texttt{latestTemp} \mapsto 73.0, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \ \texttt{currentFanState} \mapsto \texttt{Off} \\
D & \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\})
\end{array}$$

Continuing in the *Dispatch Enabled* rule, the scheduling state of `TempControl` is set to *Ready* to indicate to the underlying OS scheduler that the thread is available for scheduling.

In the `TempControl` compute entry point, dispatch is triggered by the `tempChanged` event port, which triggers the application code to read the `currentTemp` data port, store the value in the `latestTemp` thread local variable, and compare the read value to the `currentSetPoint` values. If the current temperature is greater than the high set point, a message will be put on the `fanCmd` port to turn the fan on. If the current temperature is less than the low set point, the application code puts a message into the `fanCmd` port to turn the fan off. In both cases, the application code updates the `currentFanState` variable to reflect the new desired state of the fan. Otherwise, if the current temperature is equal to or falls between the low and high set points, no command is sent and the `currentFanState` is not changed.

Picking up from the current state of the `TempControl` thread after dispatch above, in the formalization of the representation of the execution of the user code in Figure 4, the metavariables in the *ComputeEPAppCode* rule have the following values representing the application's view of the thread state at the beginning of the application code execution

$$\begin{array}{r|l}
A_t^{i} & [\texttt{currentTemp} \mapsto \langle 78.3\rangle, \texttt{tempChanged} \mapsto \langle * \rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^{o} & [\texttt{fanCmd} \mapsto \langle\rangle] \\
V_t & [\,\texttt{latestTemp} \mapsto 73.0, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \ \texttt{currentFanState} \mapsto \texttt{Off} \\
D & \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\})
\end{array}$$

and at the end of execution

$$\begin{array}{r|l}
A_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle \texttt{On} \rangle] \\
V_t^{\prime} & [\,\texttt{latestTemp} \mapsto 78.3, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \ \texttt{currentFanState} \mapsto \texttt{On}
\end{array}$$

In the context of the Figure 4 *ComputeEPInfrastructure* rule, when the above structures are placed in the context of the complete `TempControl` thread state, we have the following, representing the state of the thread after the execution of the *application code.*

$$\begin{array}{r|l}
I_t^i & [\texttt{currentTemp} \mapsto \langle 78.3 \rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^i & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle \texttt{On} \rangle] \\
I_t^o & [\texttt{fanCmd} \mapsto \langle\rangle] \\
V_t' & [\,\texttt{latestTemp} \mapsto 78.3, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \;\texttt{currentFanState} \mapsto \texttt{On} \\
D & \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\})
\end{array}$$

Then, the invocation of the SendOutput RTS moves the `fanCmd` message to the output infrastructure port state to obtain the following thread state at the completion of the execution of the *infrastructure* for the Compute Entry Point.

$$\begin{array}{r|l}
I_t^i & [\texttt{currentTemp} \mapsto \langle 78.3 \rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^i & [\texttt{currentTemp} \mapsto \langle\rangle, \texttt{tempChanged} \mapsto \langle\rangle, \\
 & \texttt{setPoint} \mapsto \langle\rangle, \texttt{fanAck} \mapsto \langle\rangle] \\
A_t^{o\prime\prime} & [\texttt{fanCmd} \mapsto \langle\rangle] \\
I_t^{o\prime} & [\texttt{fanCmd} \mapsto \langle \texttt{On} \rangle] \\
V_t' & [\,\texttt{latestTemp} \mapsto 78.3, \texttt{currentSetPoint} \mapsto (70.0, 75.0),] \\
 & \;\texttt{currentFanState} \mapsto \texttt{On} \\
D & \mathit{EventTriggered}(\texttt{tempChanged}, \{\texttt{tempChanged}, \texttt{currentTemp}\})
\end{array}$$

Following this, a director rule (omitted) resets the thread's dispatch status to *NotEnabled.* The message `On` is propagated to the `Fan` thread which is dispatched in manner similar to that illustrated above for the `TempControl` component.

10 Related Work

There are numerous contributions to the formal specification, analysis, and verification of AADL models and annexes. These works, whether implicitly or explictly, propose semantics for AADL in the model of computation and communication of the verification framework considered [6]. Many contributions focus on static semantics of models [2,34] while others consider run-time behavior and use model translation to extract executable specifications from AADL models, e.g., [4,5,10,16,37]. 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 [32], behavioral [6,35], or dependability analyses [13].

Closely related work has made strides towards the formalization of a AADL run-time behavior. Rolland et al. [31] formalized aspects of AADL's coordination and timing behavior through the translation of AADL models into the Temporal Logic of Actions (TLA+) [28]. Hugues et al. [20,29] provided a Ada/SPARK2014 definition of selected RTS as part of a broader implementation of the AADL run-time. In addition, SPARK2014 verification was used to demonstrate absence of run-time errors in the implementation. However, aspects of the semantics of the given services as well various notions of thread and system state are implicit in that they are expressed using Ada constructs (and thus rely on the underlying semantics and state of Ada). To complement this programming language-based approach, we aim for a language-independent specification of RTS and a more explicit exposition of notions of port state, thread state, as well as director and communication aspects.

11 Conclusion

Over the last fifteen years, AADL has been an effective vehicle within the formal methods community for developing rigorous model-based analysis and verification techniques. However, individual tools have often only treated a slice of AADL or have re-interpretered AADL's informal semantics to match the computional model supported by their particular tool. While it is impossible to provide a comprehensive semantics in a single conference paper, this work (with the accompanying technical report [18]) represents an initial step in providing a canonical and comprehensive treatment of AADL semantics by providing a formalization of the AADL RTS. Due to the central role played by the RTS, this formalization clarifies the standard's informal descriptions, provides a clearer foundation for model-level analysis and verification, and elucidates important aspects of the semantics for AADL code generation.

This lays the foundation for multiple lines of future work. First, AADL committee discussions suggest that future versions of the AADL will include formalizations that better ground the standard's descriptions. In addition to the RTS aspects, this will include more "coordination level" aspects that address AADL life-cycle and interactions of process and systems, scheduling, and distributed communication. This is important for establishing soundness arguments for analyses and contract languages such as AGREE [12] and BLESS [26]. Second, under consideration is a new approach to the AADL Code Generation annex which does not focus on syntactic conventions for representing AADL model artifacts but instead focuses on (a) disclosure and traceability to key aspects of a AADL RTS formalization including representation of port and thread state, implementation of AADL RTS, (b) testing- and formal-methods-based demonstration of conformance to key aspects of the forthcoming canonical semantics. Third, regarding refinement, our semantics was designed to support AADL's general threading (periodic and event-driven threads) and port communication (synchronous data flow as well as message-passing). For future work, we imagine developing refinements that specialize the semantics to particular computational paradigms (e.g., synchronous data flow only, or message passing only) – with the end result be simplified special case semantics with a proof of refinement to the more general case. Also of interest are refinements to particular scheduling regimes (our semantics focuses on the RTS and leaves a placeholder for scheduling actions that determine execution order of threads). We expect to continue refining the semantics at lower levels of abstraction to capture details of mapping AADL to classes of RTOS and standard architectures like ARINC 653 [36]. For AADL code generation that targets the formally verified seL4 micro-kernel [17,24], mechanizing the AADL semantics and formally verifying aspects AADL run-time can continue to expand the stack of formally verified infrastructure that can be applied to critical system development.

Finally, we are interested in considering issues of usability and utility for "formal semantics as documentation". In this paper, we presented formal specifications in conventional rule-based notation with symbols and rule style chosen somewhat arbitrarily. If this type of material is to be part of a industry standard used by people with a variety of backgrounds, we need formal descriptions to be presented using notations that are accessible and easy to understand. Furthermore, we need approaches for machine-readable versions of the semantics that can be leveraged by tools in multiple ways. Similar goals have existed for decades in the programming language semantics and formal methods communities. But now we find the need a bit more pressing, because to achieve the next innovations in critical system development, we need to effectively capture and connect semantics across multiple modeling languages and programming languages, along with many forms of accompanying tooling.

Acknowledgements Copyright 2022 Carnegie Mellon University, Adventium Labs and Kansas State University. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. [DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM22-0764

References

1. SAE AS5506/2. AADL annex volume 2
2. 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)
3. Barnes, J.G.: Programming in ADA. Addison-Wesley Longman Publishing Co., Inc. (1984)
4. 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)
5. 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)
6. 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)

7. Borde, E., Rahmoun, S., Cadoret, F., Pautet, L., Singhoff, F., Dissaux, P.: Architecture models refinement for fine grain timing analysis of embedded systems. In: 2014 25nd IEEE International Symposium on Rapid System Prototyping. pp. 44–50 (2014)
8. Burns, A., Wellings, A.: Analysable Real-Time Systems: Programmed in Ada. CreateSpace (2016)
9. Carré, B., Garnsworthy, J.: SPARK – an annotated Ada subset for safety-critical programming. In: Proceedings of the conference on TRI-ADA'90. pp. 392–402 (1990)
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., Amundson, I., Babar, J., Hardin, D., Slind, K., Alexander, P., Hatcliff, J., Robby, R., Klein, G., Lewis, C., et al.: Cyber-assured systems engineering at scale. IEEE Security & Privacy (01), 2–14 (2022)
12. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: NASA Formal Methods (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 (iceccs 2008). pp. 294–300 (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, Rhodes, Greece, October 17-29, 2021, Proceedings. Lecture Notes in Computer Science, vol. 13036, pp. 274–295. Springer (2021)
18. Hatcliff, J., Hugues, J., Stewart, D., Wrage, L.: Formalization of the AADL runtime services (extended version) (2021)
19. Hatcliff, J., et al.: Slang: The sireum programming language. In: International Symposium on Leveraging Applications of Formal Methods. pp. 253–273. Springer (2021)
20. Hugues, J.: A correct-by-construction aadl runtime for the ravenscar profile using spark2014. Journal of Systems Architecture p. 102376 (2022), `https://www.sciencedirect.com/science/article/pii/S1383762121002599`
21. Hugues, J., Zalila, B., Pautet, L., Kordon, F.: Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. In: IEEE International Workshop on Rapid System Prototyping. vol. 7 (2007)
22. International, S.: SAE AS5506/1, AADL Annex E: Error Model Annex. SAE International, `http://www.sae.org` (2015)
23. International, S.: SAE AS5506 Rev. C Architecture Analysis and Design Language (AADL). SAE International (2017)
24. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: seL4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220 (2009)

25. Larson, B., Hatcliff, J., Fowler, K., Delange, J.: Illustrating the AADL error modeling annex (v.2) using a simple safety-critical medical device. In: Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology. pp. 65–84. HILT '13, ACM, New York, NY (2013)
26. 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)
27. Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina : An environment for AADL models analysis and automatic code generation for high integrity applications. In: Reliable Software Technologies – Ada-Europe 2009. pp. 237–250. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
28. Merz, S.: The specification language TLA+. In: Logics of specification languages, pp. 401–451. Springer (2008)
29. Mkaouar, H., Zalila, B., Hugues, J., Jmaiel, M.: A formal approach to AADL model-based software engineering. International Journal on Software Tools for Technology Transfer **22**(2), 219–247 (2020)
30. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T.: TASTE: A real-time software engineering tool-chain overview, status, and future. pp. 26–37 (01 2011)
31. 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)
32. Sokolsky, O., Lee, I., Clarke, D.: Schedulability analysis of aadl models. In: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium. pp. 8–pp. IEEE (2006)
33. Stewart, D., Liu, J.J., Cofer, D., Heimdahl, M., Whalen, M.W., Peterson, M.: AADL-based safety analysis using formal methods applied to aircraft digital systems. Reliability Engineering & System Safety **213**, 107649 (2021)
34. 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 (2020)
35. Tan, Y., Zhao, Y., Ma, D., Zhang, X.: A comprehensive formalization of AADL with behavior annex. Scientific Programming **2022** (2022)
36. VanderLeest, S.H.: ARINC 653 hypervisor. In: 29th Digital Avionics Systems Conference. pp. 5–E. IEEE (2010)
37. 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)
38. Yuan, C., Wu, K., Chen, G., Mo, Y.: An automatic transformation method from AADL reliability model to CTMC. In: 2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE). pp. 322–326 (2021). https://doi.org/10.1109/ICICSE52190.2021.9404135