## Overview

These instructions assume that you have read through the HAMR Overview (ch00-hamr-overview.html#ch00-hamr-overview) and HAMR Tour (choX-hamr-tour.html#chox-hamr-tour) material.

HAMR supports model-based system development using multiple programming languages including Slang and C. Systems can be created for execution on multiple platforms. The HAMR tool chain itself can be installed on Windows, Linux, and MacOS platforms.

The programming language and system execution platform that you choose determines the tools that you will work with and the workflows that you will follow.

The HAMR Workflows video and slides on the HAMR Videos (../videos.html#videos) page illustrate possible HAMR workflows and installation options.

In general, using HAMR to build an executable system requires working with:

- The Eclipse-based OSATE or OSATE customization FMIDE (Formal Methods Integrated Development Environment) for creating AADL models and invoking HAMR code generation. We recommend using the FMIDE because the HAMR plug-in is already installed in it. If you are using AADL for other purposes and already have OSATE installed (perhaps with other plug-ins added), you can install the HAMR plug-in directly.
- A development environment for the HAMR targeted programming language(s) – the programming language used to code the implementation of components whose skeletons and associated run-time infrastructure are generated by HAMR. HAMR currently supports Slang (a Scala-based language), C, with some support for CakeML. Currently, HAMR support for Slang is the most mature and easy-to-use. When coding components in Slang, you will need to install the Sireum IVE (Integrated Verification Environment) customization of the IntelliJ IDE – which will support you in coding/debugging Slang with extensions in Scala and Java. When coding components in C, you will need to install a C IDE that can process CMake files. We recommend using the JetBrains CLion IDE. However, other IDEs such as Microsoft Visual Studio Code can also be used.
- Execution platform support - HAMR systems can be built for JVM, Linux/Mac OSX based operatoring systems, and the seL4 microkernel running in the Qemu simulator or on a development board.

There are at least two approaches to installing the tool chain elements above:

- install them directly, with assistance from Sireum Kekinian framework. Kekinian is installed first, and then a few simple command line options can be used to install the FMIDE, Sireum IVE, JVM, etc. This is recommended when you are working with Slang and JVM deployments, and it is also suitable for C-based development on Linux.

- install them via a virtual machine built using the Vagrant VM build tool - As part of the DARPA CASE project, Vagrant scripts (https://github.com/sireum/case-env) are provided to build a VirtualBox VM running Debian Linux that hosts the tool components above. If you are working with seL4 or if you are working with DARPA CASE technologies and examples, it is strongly recommended that you install the HAMR tool chain using the Vagrant VM scripts.

The following describes the most common HAMR workflows and recommended installation approaches. These workflows are described in greater detailed in the video and slides available on the HAMR Videos (../videos.html#videos) page.

- **Component development using the Slang programming language with system deployment on the JVM -** Tools used include the Eclipse-based OSATE or FMIDE (OSATE customization) for creating AADL models, Sireum IVE (IntelliJ customization) for coding and running Slang. The workflow can be set up on Mac OSX, Linux, or Windows. The preferred way to install is to download the Sireum Kekinian framework and use it to install the necessary components. Alternatively, you can use the DARPA CASE Vagrant VM building infrastructure to build a virtual machine running Debian with all the required tools installed.
- **Component development using the C programming language with system deployment on Linux/MacOS X -** Tools used include the Eclipse-based OSATE or FMIDE (OSATE customization) for creating AADL models, and CLion IDE for coding and running C (other C IDEs can be used as well). The preferred way to install is to use the DARPA CASE Vagrant VM building infrastructure to build a virtual machine running Debian with all the required tools installed. Alternatively, you can download Sireum Kekinian and use it to install the required tools directly in Linux or Mac OSX.
- **Component development using the C programming language with system deployment on seL4 micro-kernel -** Tools used include the Eclipse-based OSATE or FMIDE (OSATE customization) for creating AADL models, and CLion IDE for coding and running C (other C IDEs can be used as well), and multiple components from Data61 for working with seL4. The preferred way to install is to use the DARPA CASE Vagrant VM building infrastructure to build a virtual machine running Debian with all the required tools installed. Alternatively, you can download Sireum Kekinian and use it to install the required tools directly in Linux or Mac OSX. You will also need to download and install the Data61 tools for seL4.

Following the installation instructions, instructions are given for loading and running a simple HAMR Slang project.

# Installation Option: Direct Installation with Assistance from the Sireum Kekinian Framework

## Installing and Updating FMIDE (customization of AADL OSATE IDE) and HAMR

You will first install SAnToS Lab Sireum Kekinian programming language tools, and then use that to install the FMIDE for creating AADL models and performing HAMR code generation.

- **Windows:**

```
git clone https://github.com/sireum/kekinian Sireum
cd Sireum
git submodule update --init --recursive
bin\build.cmd setup
bin\install\fmide.cmd
start /B bin\win\fmide\fmide.exe
```

- **Linux:**

```
git clone https://github.com/sireum/kekinian Sireum
cd Sireum
git submodule update --init --recursive
bin/build.cmd setup
bin/install/fmide.cmd
bin/linux/fmide/fmide&
```

- **macOS:**

```
git clone https://github.com/sireum/kekinian Sireum
cd Sireum
git submodule update --init --recursive
bin/build.cmd setup
bin/install/fmide.cmd
open bin/mac/fmide.app
```

The Sireum directory created above is referred to using the environment variable `$SIREUM_HOME` in the instructions below.

Occassionally, the FMIDE will be updated to include new HAMR code generation or AADL analysis capabilities. To update the FMIDE, simply re-run the FMIDE install script as illustrated below.

- **Windows:**

```
%SIREUM_HOME%\bin\install\fmide.cmd
```

- **Linux/macOS:**

```
$SIREUM_HOME/bin/install/fmide.cmd
```

If the installation somehow did not finish (e.g., due to a network issue), remove the problematic file in Sireum's cache directory (~/Downloads/sireum) and re-run the above.

Updates to the Sireum framework may add new code-level verification, analysis, and compilation functionality. When you update Sireum, you should update the FMIDE as well. To update Sireum and the FMIDE, with your current directory as `$SIREUM_HOME` , simply do a git pull with –recurse-submodules option, and re-run build.cmd setup and fmide.cmd, e.g., for macOS this would be

```
git pull --recurse-submodules
bin/build.cmd setup
bin/install/fmide.cmd
```

Note that it is also possible to download specific versions (specific git commit tips) for both Sireum and the FMIDE. For details, see https://github.com/sireum/case-env (https://github.com/sireum/case-env).

## IDE (Sireum IVE) for Slang-based Development of HAMR Systems

When using Slang to program an AADL/HAMR based system, you will use the Sireum IVE (Integrated Verification Environment). Sireum IVE is a customization of IntelliJ that includes Scala-associated infrastructure pre-installed as well as other SAnToS-built plug-ins for the Slang language subset and associated analysis/verification capabilities.

The Sireum Kekinian framework is used to install and update the Sireum IVE. Thus, Sireum IVE is installed as part of the steps above.

Then, to launch it:

- **Windows:**

```
start /B %SIREUM_HOME%\bin\win\idea\bin\IVE.exe
```

- **Linux:**

```
$SIREUM_HOME/bin/linux/idea/bin/IVE.sh&
```

- **macOS:**

```
open $SIREUM_HOME/bin/mac/idea/IVE.app
```

## IDE for C-based Development of HAMR Systems

Any C modern IDE can be used. The HAMR team uses CLion from JetBrains.

To install and run CLion C/C++ IDE (license required/free 30-day evaluation):

- **Windows:**

```
%SIREUM_HOME%\bin\install\clion.cmd
start /b %SIREUM_HOME%\bin\win\clion\bin\clion64.exe
```

- **Linux:**

```
$SIREUM_HOME/bin/install/clion.cmd
$SIREUM_HOME/bin/linux/clion/bin/clion.sh&
```

- **macsOS:**

```
$SIREUM_HOME/bin/install/clion.cmd
open $SIREUM_HOME/bin/mac/clion/CLion.app
```

# Installation Option: Vagrant VM-based Installation

Follow the instructions from DARPA CASE Github repository. Vagrant scripts (https://github.com/sireum/case-env) are provided to build a VirtualBox VM running Debian Linux that hosts the tool components above as well as tools supporting development for seL4. The VM also includes other CASE analysis and model transformation tools.

- General Concepts
- Supported Programming Languages for Application Logic
- Supported Target Platforms
    - Java Virtual Machine (JVM)
    - Linux (xNix)
    - seL4 Microkernel
    - Variations and Extensions
- HAMR on the DARPA CASE Program
- HAMR Code Generation Architecture
- Assurance Case Structure of HAMR-built Systems

## General Concepts

HAMR ("[H]igh [A]ssurance [M]odeling and [R]apid engineering for embedded systems") is a code generation and system build framework for cyber-resilient systems whose architecture is specified using the Architecture Analysis and Design Language (https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=191439) (AADL). HAMR supports development of new components and wrapping of legacy components by generating code that provides interfacing infrastructure between components. Once component implementations are developed, HAMR can create deployable builds on multiple platforms including the Java Virtual Machine, Linux, and the seL4 micro-kernel (https://sel4.systems).

For interface and infrastructure generation, HAMR translates AADL system models to code that implements threading infrastructure and inter-component communication for AADL components. HAMR *does not* generate code for the application logic of a system; it only generates infrastructure code for "gluing together" the component application code. So to "program a system" using HAMR, you define a component-based system/software architecture in AADL, you program the internal behavior (the "application code") of the components in a programming language (HAMR supports several different languages, plus some high-level declaration approaches for generating message filters for cyber-resiliency) and then HAMR generates the infrastructure code (aligned with the AADL architecture) that provides an execution context for the application code.

The auto-generated infrastructure code is designed to implement the semantics of AADL threading and communication as specified in the AADL standard (https://www.sae.org/standards/content/as5506c/) (HAMR implements a subset of the standard semantics). In essence, AADL defines a *computational model* – a constrained way of organizing processes, threads, and communication – that matches the domain properties of real-time embedded systems. AADL and its computational model are system engineering abstractions that are designed to be analyzable. Since the HAMR auto-generated infrastructure code is faithful to the AADL semantics and computational model, various forms of analysis applied to AADL model are faithful to the semantics and resiliency properties of the executable code in the HAMR-generated deployed system.

An important aspect of the HAMR tool chain is that it enables the application code that implements a component to be platform agnostic to a significant degree. Application programmers program to a common collection of APIs for threading and communication aspects auto-generated by HAMR from AADL. For example, for each AADL component, a collection of APIs is auto-generated to support communication over the ports appearing in the correspondign AADL component interface. These APIs hide the details of how threading and communication are realized on particular platform – regardless of the particular platform, the *behavior* of the threading and communication infrastructure should be the same, even though the *implementation* differs.

Use of the AADL-based threading and communication abstractions helps system designers and application programmers develop a consistent and shared understanding of how to organize computation and communication within the system. In addition, they simplify integration efforts because all communication is based on a fixed collection of interaction patterns, and all threading is based on analyzeable approaches inspired by decades of experience in the real-time embedded systems community. Note that platform specific I/O interfacing to sensors/actuators, services etc. may still need to be reworked when moving to another platform.

**Note**

When a new platform support is added to HAMR, it is the responsibility of the HAMR translation developer to ensure that the semantics of the new platform infrastructure code supports the intended semantics of the APIs, as indicated by the AADL standard. The AADL standard is not completely clear about some of the intended behavior. At this time there is no formal process within the AADL community to establish compliance to the standard. We are addressing these gaps in ongoing research. Currently, HAMR takes the following approach: (a) a reference implementation for the infrastructure semantics (AADL computational model) is programmed in Slang (Slang is described in the section below), and this reference implementation can be executed on the Java Virtual Machine, and (b) initial infrastructure is being developed to enable HAMR platform implementations to compare testing results to results of the reference implementation (in essence, this is a model-based testing approach where there high-level Slang-based reference implementation is the "model" that serves as the test oracle). A formal semantics of the AADL computational model is also under development, and as of now, the HAMR assurance approach includes manually comparing the implementation of the Slang reference implementation to formal semantics to determine conformity. Longer-term research goals include formal proofs of correspondence between a mechanized formalization of the AADL semantics (e.g., in Coq), HAMR Slang reference implementation, and back-end implementations.

## Supported Programming Languages for Application Logic

The application code (i.e., business logic) that provides the functionality of AADL components can be written in one of several languages including:

- C – the most widely used language for embedded system programming,
- Slang – a safety-critical subset of Scala with automated contract-based verification support and ability to compile to C, developed by researchers in the SAnToS Lab research group at Kansas State University, and
- CakeML – a language with verified machine code generation and accompanying proof tools that enable applications to be proved correct using theorem-proving technology.

Experimental prototypes have also been developed for programming component application logic using the Behavior Annex (BA) of AADL and the variant of BA supported by the BLESS verification framework – BA and the BLESS BA variant are state transition system notations which have both textual and graphical representations.

Regardless of the language used, the business logic and associated executable code is held in a location known to HAMR (via AADL model properties or HAMR configuration information). HAMR takes the user-written application code and weaves it together with the auto-generated infrastructure code to form a system build.

## Supported Target Platforms

HAMR supports code generation and system builds for several platforms.

### Java Virtual Machine (JVM)

HAMR generates infrastructure code in Scala - a widely used cross-paradigm language (e.g., supporting both OO and functional programming) that is widely used in the enterprise domain. Scala is compatible with Java (Scala code can easily Java code, and vice versa) and Scala code is compiled to JVM. Scala can also be compiled directly to native code. When HAMR generates Scala infrastructure code, application component logic can be programmed in Scala or Slang (or Java, with a bit of additional interface engineering).

Much of the Scala AADL infrastructure code generated by HAMR is in the Slang high-assurance language subset. Not only is Slang designed for verifiabilty, it has a restricted memory model that facilitates compilation to C with static memory layout and other properties relevant for embedded systems. HAMR's C-based xNix backends are largely generated from this Slang. This enables a significant amount AADL infrastructure implementation to be shared across the JVM, xNIX, and seL4 platforms. This architecture is designed to aid future assurance efforts.

In general, the benefit of the HAMR Slang code generation combined with Scala/Slang application code is that a system can easily be simulated, tested, and dynamic behavior visualized in a variety of ways using a JVM-based simulation framework provided by HAMR. This enables developers to rapidly prototype a mockup of a system and organize and debug component interfaces/integration. Subsequently, the complete system can be programmed in Slang, verified using Slang / Logika verification framework, and, if desired, compiled to C for deployment. The C compiled from Slang can be deployed to x-nix (i.e., Linux, MacOS) or the seL4 platform infrastructure described below.

Since JVM-based HAMR builds rely on the JVM for threading and communication (e.g., instead of an RTOS) the real-time behavior constraints of AADL are not guaranteed to be met. The general approach is to use the JVM-based build to prototype system structure, interfaces, and non-real-time related functional behavior.

*Use of Slang, Scala, and the JVM-based framework are not required when using HAMR*. One can have a development workflow totally based on C using the platform backends described below.

## Linux (xNix)

HAMR can generate infrastructure code and builds in C suitable for deployment on Linux and macOS (we refer to these collectively as the xNix platform). In the current xNix platform builds, xNix processes are used to realize AADL thread components and Unix System V-based communication primitives are used realize AADL intercomponent communication. To support C-oriented workflows, HAMR generates APIs for component application logic in C, and a C development environment of choice can be used to develop and debug component implementations.

Since Linux and macOS are not real-time operating systems, the real-time threading and communication behavioral constraints of AADL are not guaranteed to be met. Also, Linux and macOS do not provide the same level of memory protection that one gets with, e.g., the seL4 microkernel. Nevertheless, the HAMR generated code for these platforms organizes the system build into distinct units with constrained interactions between units. Even without the strong OS/platform partitioning, this disciplined build structure can aid assurance arguments.

Sometimes, HAMR's support for xNix is in conjunction with its support for seL4: HAMR's support for Linux and Linux in VMs facilitates wrapping of legacy components and subsystems (perhaps untrusted) and isolating them in a VM in a seL4 partition.

## seL4 Microkernel

The seL4 verified micro-kernel (https://sel4.systems) is a key technology solution used to provide strong partitioning between components in a application. With this partitioning, one application component cannot access the memory/state of another component unless explicitly granted that capability in the kernel

configuration, and communication between components is guaranteed to be limited to explicitly declared pathways. Based on these partitioning properties, seL4 can play a key role in achieving fault containment and cyber-resiliency by

- isolating less trusted portions of the application in separate components so that they cannot attack or interfere with critical components,
- separating trusted components to better isolate faults and achieve other robustness properties, and
- enabling communication between components to be more easily assessed/audited and guarded with various security controls.

HAMR-generates C APIs for component logic application logic that are identical to those generated for xNix. This enables systems to be prototyped in xNix (which has richer support for debugging and drivers) and then moved to seL4. The API-based isolation of the application code from the details of the underlying platform also enables general portability between platforms.

When HAMR generates implementations for seL4, it generates realizations of AADL component interfaces and connection topologies in CAmkES (https://docs.sel4.systems/projects/camkes/) – a dedicated language from Data61 for specifying seL4-based partitions and inter-partition communication. Once all component implementations and other build artifacts are in place, HAMR invokes a Data61-provided translation tool that translates CAmkES descriptions to C and seL4 capability configuration files that control the memory protection mechanisms of the seL4 microkernel. Additional Data61 tools generate the binaries that are deployed on seL4.

### Variations and Extensions

HAMR is designed to make it easier to target multiple platforms and communication infrastructure, even within the same system. While the current HAMR implementation only supports the platforms above, future research efforts may add support for widely-used distributed communication frameworks like OMG'S Data Distribution Service (DDS), ActiveMQ, zeroMQ, etc. Support for real-time operating systems and micro-kernels with strong temporal partitioning is also planned (simple HAMR extensions have been prototyped to support FreeRTOS).

## HAMR on the DARPA CASE Program

The Collins Aerospace development tools for the DARPA CASE program (http://loonwerks.com/projects/case.html) emphasize AADL to C workflows. In these workflows, HAMR auto-generates C-based infrastructure code for AADL-compliant threading and communication that can be deployed on Linux or "bare-metal" seL4. HAMR is used to generate C-based APIs for interacting over AADL ports, and AADL component application logic is written in C.

CASE program scope *does not* include full support for enforcement of temporal separation as one would get from an RTOS or from the emerging mixed-criticality (MCS) version of seL4. However, CASE AADL modeling guidelines and HAMR code generation are oriented to move in that direction, e.g., by embracing a simplified version of ARINC 653 threading and communication and be taking advantage of existing domain scheduling capabilities of the seL4 microkernel.

In addition to C based development, the Collins Aerospace solutions use CakeML for selected components that benefit from higher levels of assurance. For example, for CakeML implementations are used for the following types of cyber-resiliency components:

- *filters* - components that filter out problematic intercomponent messages as might be generated by an adversary,
- *monitors* - components that monitor system state and communication and then generate alerts or perform recovery actions, and

- *an attestestation framework* - infrastructure that can automatically measure various attributes of a component to confirm its authenticity.

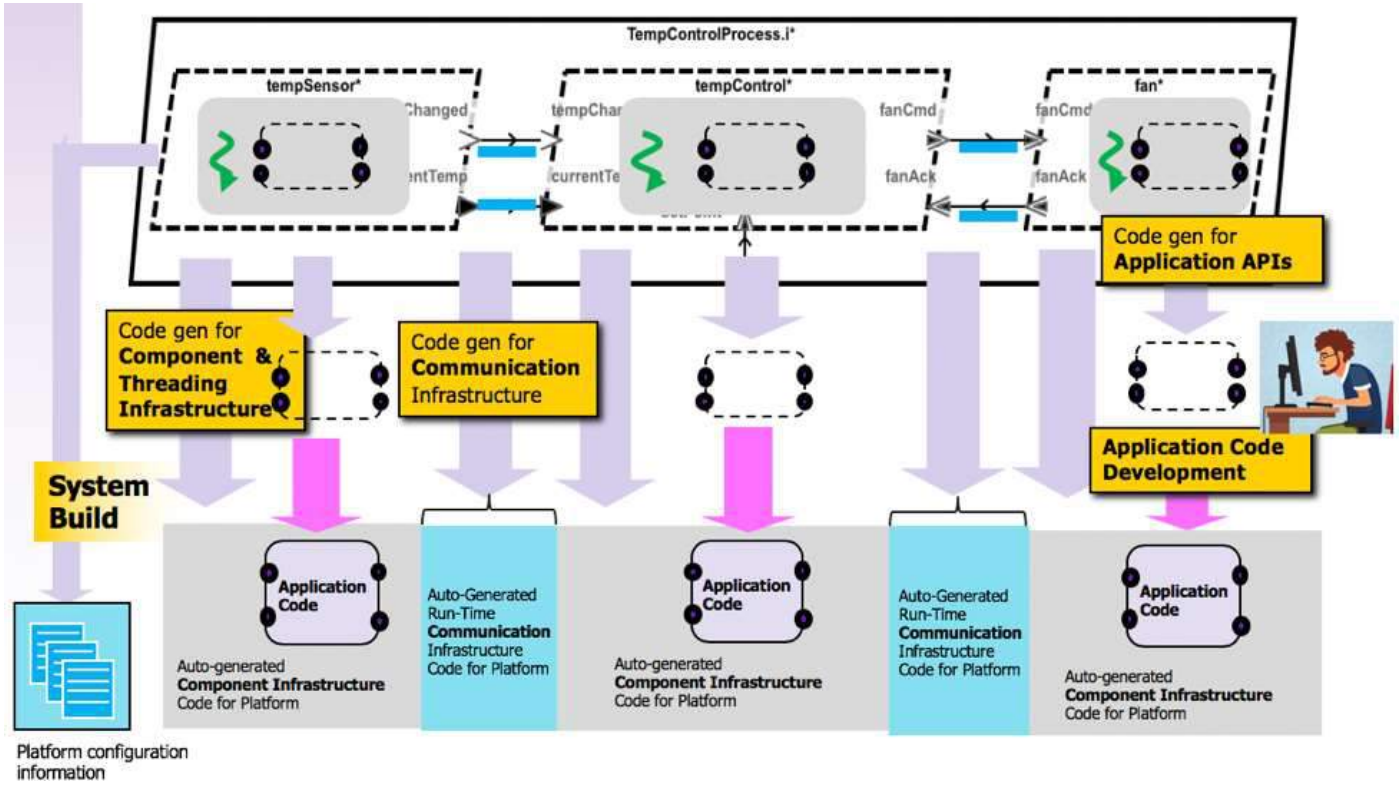
For these types of components, CakeML is auto-generated from higher-level formal specifications such as extended regular expressions (for filters) and variants of temporal logic (for monitors). The ultimate goal of these efforts (not all of which can be achieved in CASE scope), is to have machine-verified proofs that a deployment cyber-resiliency component's implementation satisfies its formal high-level specification.

Slang is used in Collins Aerospace research for system prototyping. Moreover, "behind the scenes", the generation of C code for platform infrastructure is factored through Slang in many cases (i.e., code is first generated for Slang, and then the Slang is compiled to C) which enables the tooling to more easily target multiple backends. The factoring through Slang is invisible to C-oriented application developers.

On CASE, HAMR's cross-platform build capabilities are used primarily to help migrate legacy Linux applications to seL4. For example, HAMR can generate seL4/CAmkES partitions that host virtual machines (VMs) running legacy applications in Linux. Moreover, HAMR's auto-generated APIs for communication make it easier to move implementations between platforms (e.g., shifting code running in a Linux process to code running on bare-metal in seL4 CAmkES component – as might be necessary when extracting key components out of an untrusted Linux-based deployment and hardening them to trusted components to run in an seL4 partition).

Overall, in the CASE Collins Aerospace tools, HAMR forms the backbone of tool-chain for engineering cyber-resilient embedded systems by supporting modeling, analyzing, transforming, code generation, and build construction for partitioned applications.

## HAMR Code Generation Architecture




**HAMR Code Generation Architecture**

HAMR Code Generation Architecture presents the high-level concept of the HAMR code generation. The AADL model in the figure is a simple temperature control system with a temperature sensor, a controller (thermostat), and a cooling fan – the details are not relevant for this overview.

A fundamental goal of HAMR is to structure the code so that the application code is isolated from the underlying component and communication infrastructure – enabling HAMR to target platforms with different threading and communication with minimal changes to application code.

To achieve this, the code generation has the following four dimensions.

- **Code skeletons and APIs for application code** – for each AADL thread, HAMR generates code skeletons for application code (see right hand side of Figure 28– forthcoming DARPA CASE releases will document the details of these code skeletons). The developer completes the component implementation using a conventional development environment. The application code accesses automatically generated APIs to communicate via the component's ports with other components. These APIs abstract the application code away from the details of the particular communication infrastructure.
- **Component infrastructure, including threading infrastructure** – code is generated to support the invocation of the application code and move the data being transported over the component's ports between port variables accessible by the application code and the communication infrastructure abstracted by the communication APIs.
- **Communication infrastructure** – code is generated to move data/events between output ports of sending components and input ports of receiving components. The particular mechanism used depends on the underlying platform. For example, for a Unix/Linux platform-based implementation, System V communication primitives used. For an seL4-based implementation, seL4's inter-partition communication primitives are used. For distributed components, a publish-subscribe framework like OMG Data Distribution Service or a CAN BUS could be used (distributed communication is not yet implemented in HAMR).
- **Platform configuration information -** Depending on how the underlying platform capabilities are configured, HAMR may generate meta-data (e.g., in an XML or JSON format) that is used to configure options on the underlying platform.

## Assurance Case Structure of HAMR-built Systems

Overall, HAMR can be seen as coordinating an AADL-guided build of a system that includes the four dimensions above. The overall assurance case that the generated build achieves desired system functional/safety/security requirements (including cyber-resiliency requirements) includes the following elements:

- **(Arch) -** arguments that an architecture appropriate for the system requirements (e.g., achieving appropriate partitioning properties) is specified in AADL (reflected in the topological structure of the AADL specification),
- **(ArchAttributes) -** arguments that system attributes presented (e.g. as AADL properties) are appropriate for the system requirements,
- **(ArchAnalysis) -** arguments that any automated analyses including information flow analysis, latency analysis, schedulability analysis, etc., that are used to automatically configure model attributes or to confirm that model attributes appropriately realize system requirements,
- **(ApplicationCode) -** arguments each component's application code is implemented to satisfy component behavioral specification so that integration of the components with component/communication infrastructure that adheres to the AADL computational model yields "end to end" behavior that achieves the system requirements,
- **(ComponentInfrastructureCode) -** arguments that the HAMR code generation for component infrastructure correctly implements the AADL computational model,
- **(CommunicationInfrastructureCode) -** arguments that the HAMR code generation for communication infrastructure correctly implements the AADL computational model,

- **(PlatformConfigurationInformation) -** arguments that the HAMR code generation for platform configuration information configures the platform to achieve resource provisioning/partitioning and other properties necessary to realize the system requirements.

- Overview
- AADL Modeling
    - The OSATE Eclipse AADL Environment
    - Component Modeling
    - Defining Data Types for Intercomponent Communication (shared data, messages)
    - System Modeling
- AADL Analysis
    - Information Flow Analysis
- Slang Workflows
    - Generating Code and Code Development Environment Setup in the IntelliJ-based Sireum Integrated Verification Environment (Sireum IVE)
    - Build Scripts
    - Component Development
    - System Development and Execution
    - Unit Testing
    - System Testing
    - System Monitoring and Execution Visualization
- C Workflows
    - Generating Code and Code Development Environment Setup
    - Component Development
    - Scheduling and System Configuration
    - Linux System Execution
    - seL4 System Execution (simulation via Qemu)
    - seL4 System Execution (on development board)

## Overview

This chapter provides a high-level summary of developer activities when using HAMR. This includes:

- simple illustrations of AADL component and system modeling,
- examples of HAMR-related AADL model analysis,
- generating code with HAMR,
- programming component behaviors with HAMR,
- HAMR-based testing for components (unit testing) and systems (system tests), and
- running HAMR applications.

HAMR supports multiple programming languages for coding the functionality of components and systems whose infrastructured is auto-generated from HAMR. Workflows for both Slang and C programming languages are illustrated in this chapter. The first two items above (AADL modeling and AADL analysis) are independent of whether or not Slang or C is being used as the programming language. Accordingly, the content of the chapter is organized as follows:

- AADL modeling
- AADL analysis
- Slang workflows
- C workflows

Throughout the presentation, forward references are provided to subsequent chapters that provide more details.

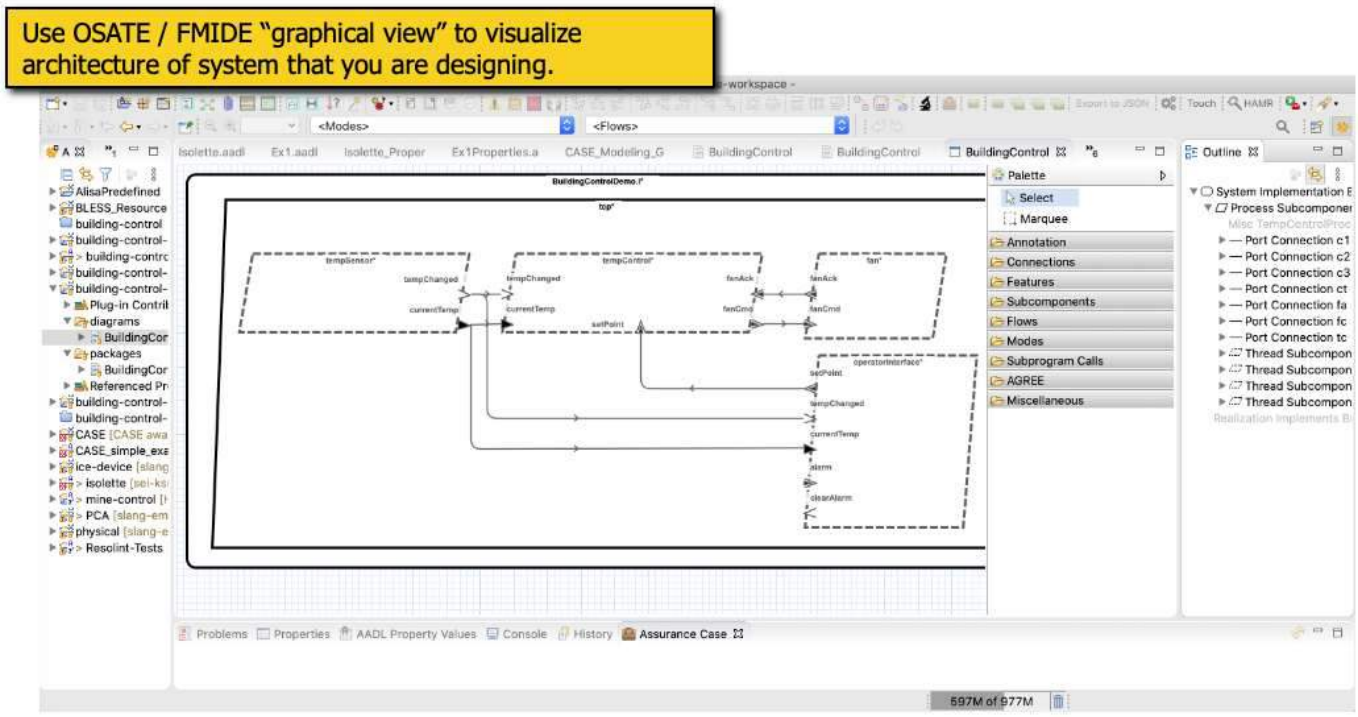
# AADL Modeling

## The OSATE Eclipse AADL Environment

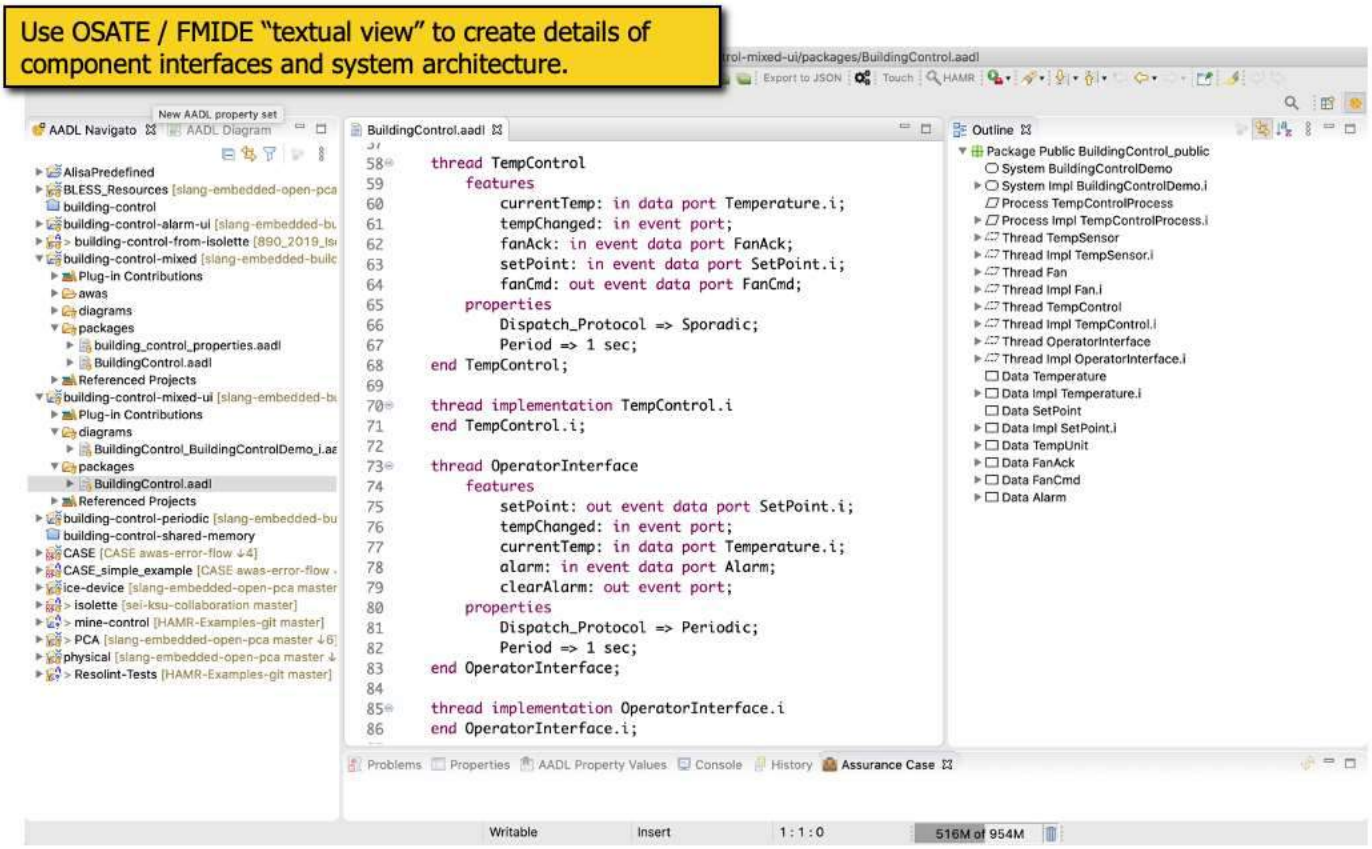
HAMR workflows begin with creating AADL models of components and systems in the *OSATE* AADL tool. OSATE is based on the widely-used open-source Eclipse IDE. HAMR is currently distributed in special "branded" distribution of OSATE called the Formal Methods Integrated Development Environment (FMIDE) developed on the DARPA CASE project.

- See the installation instructions for installing OSATE/FMIDE and HAMR plug-in.
- See AADL Computational Paradigm and HAMR Fundamental Concepts (ch04-hamr-aadl-fundamental-concepts.html#ch04-hamr-aadl-fundamental-concepts) for a detailed presentation of AADL modeling and semantics concepts.

Each OSATE AADL model has both a graphical view and textual view, and OSATE keeps them synchronized. Although it is possble to create and edit models in the graphical view, HAMR documentation emphasizes creating/editing models in the textual view and then visualizing them in the graphical view.

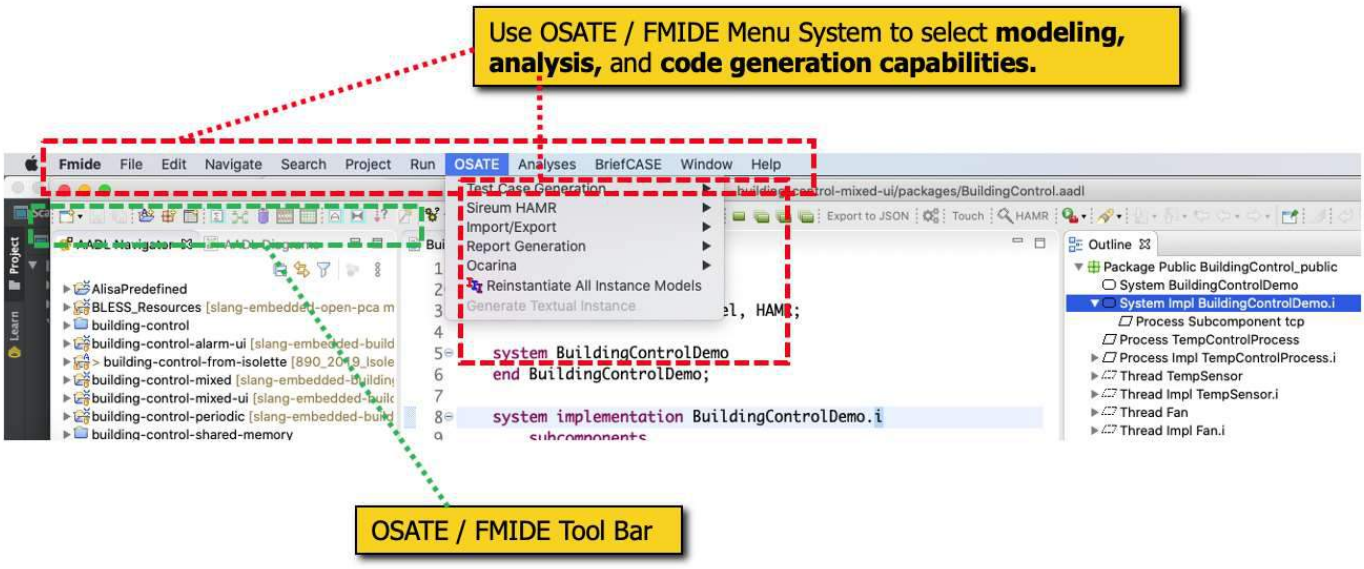



**OSATE AADL Graphical View**




**OSATE AADL Textual View**

OSATE includes different analysis and code generation plug-ins whose functions can be accessed through the menu system or through icons in the tool bar. Some OSATE/FMIDE capabilities require additional libraries to be installed, they vary in their maturity, and they work on different model features/properties (some of which may not be included in your models) – so the take away is that you should not expect all other OSATE plug-ins to work on models in the HAMR documentation nor models that you build. However, all HAMR-related related features described in this documentation should work.




**OSATE Menus and Toolbar**

## Component Modeling

AADL systems are built from different kinds of components.

With HAMR, the most important component kind is a `thread` component, which corresponds to a *real-time task* in real-time systems literature.

HAMR generates code skeletons for `thread` components, and a developer implements a system by (a) using a HAMR-supported programming language to program the behavior of thread components and (b) using other AADL modeling elements to group and "wire" components together.

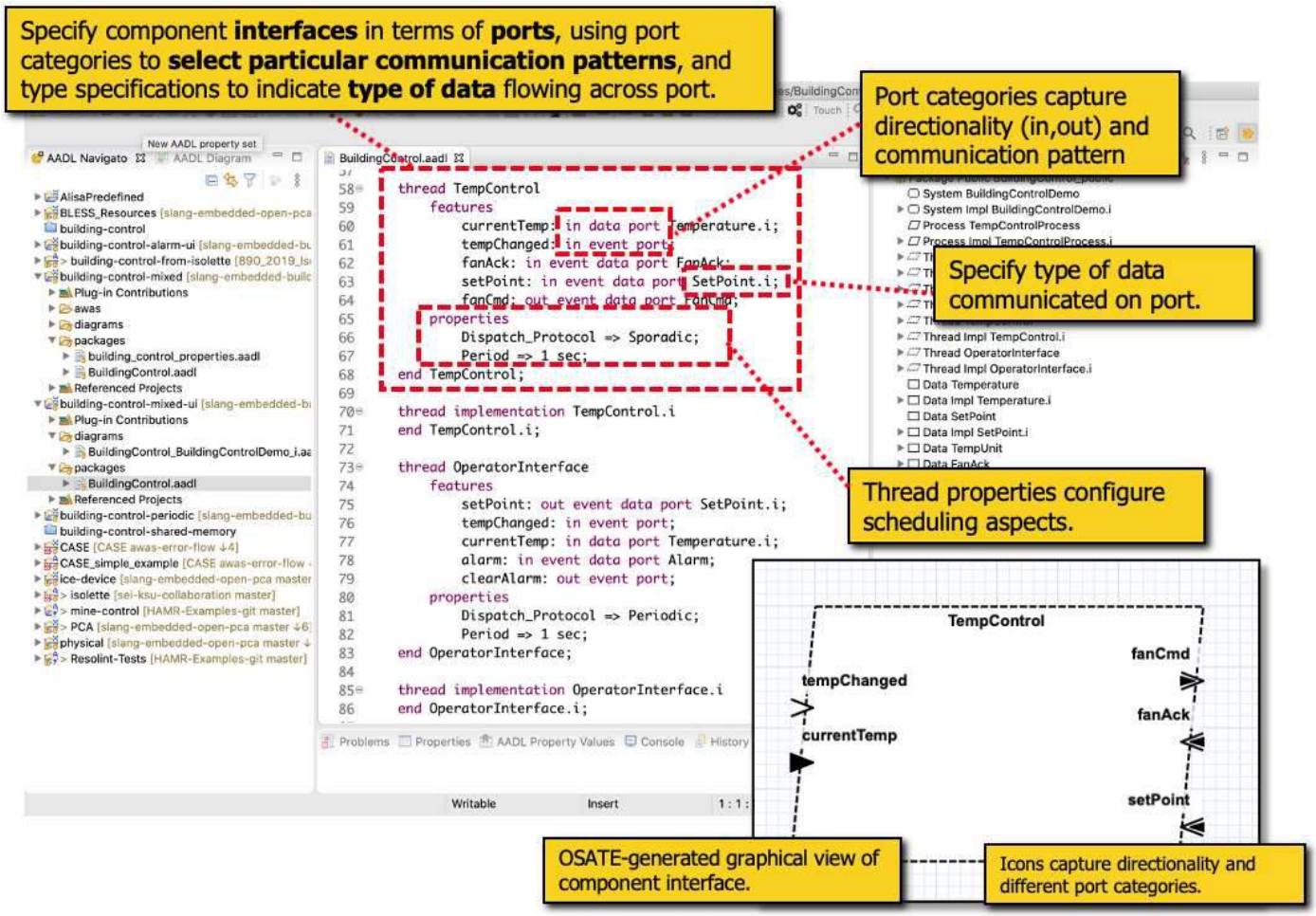
HAMR also generates for communication infrastructure for the wiring between components, and platform infrastructure code corresponding to other kinds of AADL components such as `process` and `system` – which are used to aggregate `thread` components so that they can be associated with various system resources and configurations.

Each AADL software component, e.g., a `thread`, `process`, or `system`, has a specified interface captured as an AADL `component type`. AADL emphasizes *port-based inter-component communication*, and so an AADL component type (interface) is a collection of ports. There are several categories of AADL ports (`event`, `data`, and `event data`), and a port category (together with optional port properties) indicates a particular pattern of communication to be used for communication on that port.

- `data` ports – roughly correspond to shared memory between components that act either as writers or readers
- `event` ports – roughly corespond to buffered notifications from one component to another
- `event data` ports – roughly correspond to buffered messages with payloads sent from one component to another

AADL properties can be used to configure buffer sizes, timing properties, etc.

HAMR only supports **uni-directional** ports and connections (with port direction indicated by `in` and `out`), so as to enable (a) clean design of security and information flow controls and (b) model- and code-level information flow analysis.



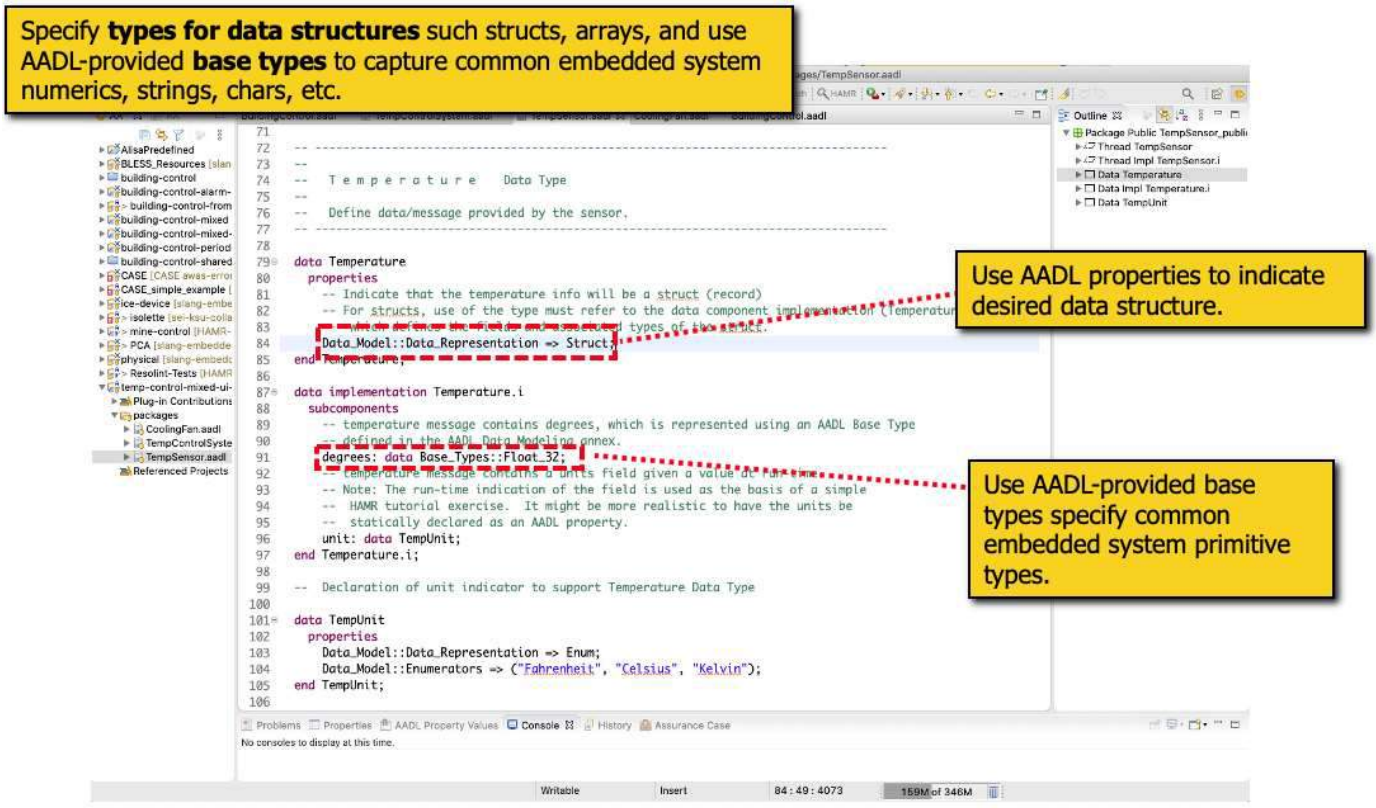

**AADL Textual Representation of Port-based Thread Component Type (Interface)**

Later in the workflow, HAMR will generate code-level interfaces for modeled component types.

## Defining Data Types for Intercomponent Communication (shared data, messages)

AADL provides facilities modeling data structures like structs (records), arrays, unions, and enums that flow between components. AADL libraries also provide common base types including fixed-width, signed and unsigned integers, floats, etc. commonly used in embedded system programming.

Once these types are defined, they are referenced in component interface specification to indicate the types of data flowing across ports.



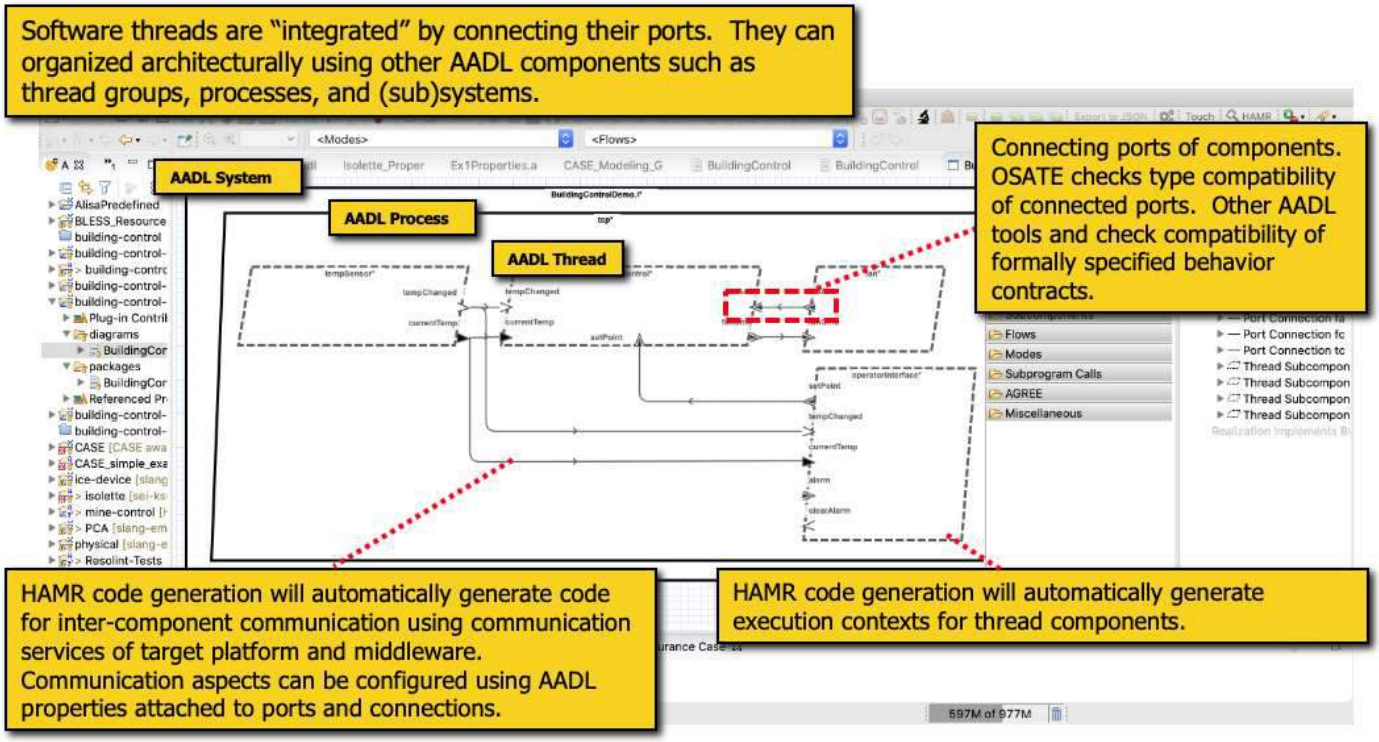

**AADL Data Modeling**

Later in the workflow, HAMR will generate code-level data types for modeled component types.

## System Modeling

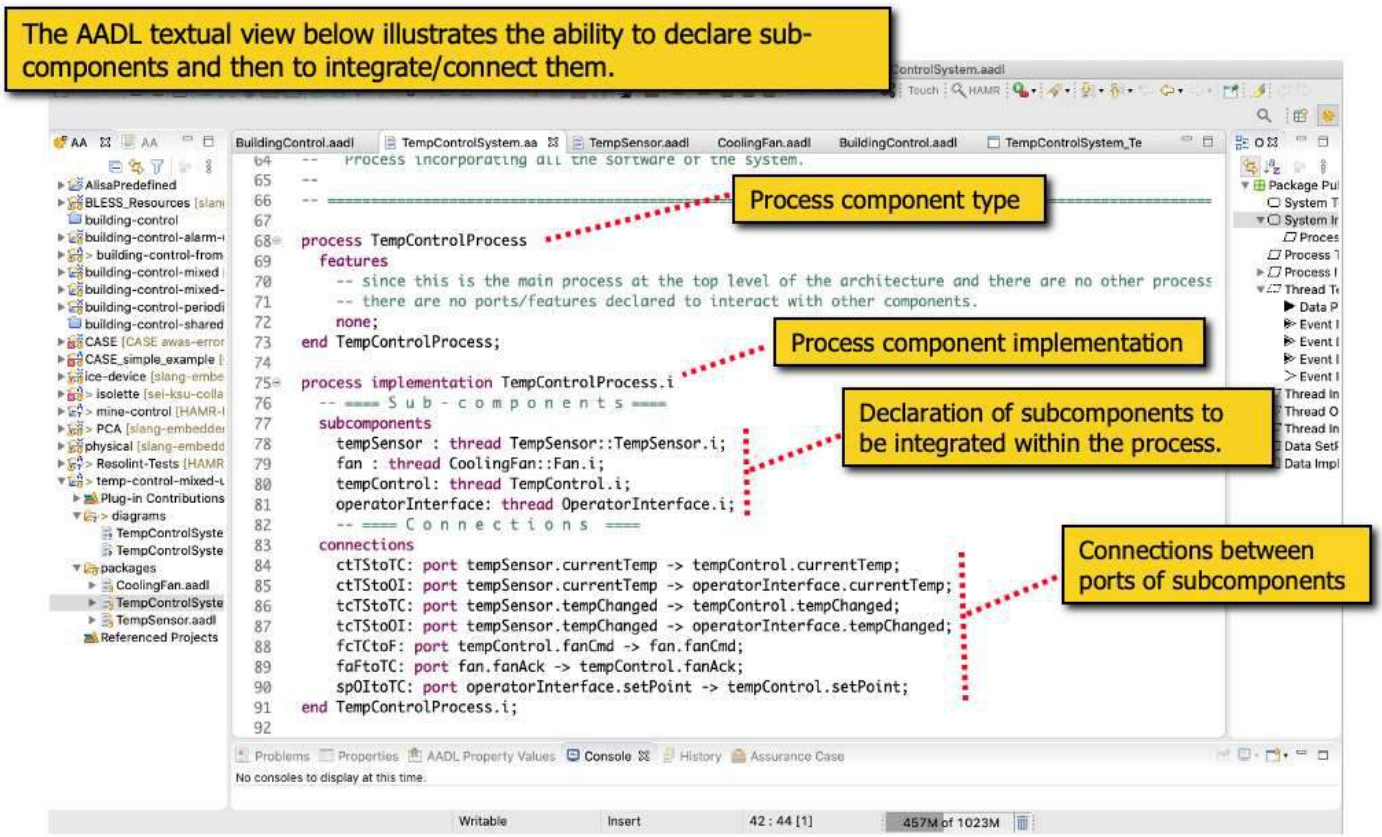
### Software Integration and Architecture

Once software threads are designed, they can be integrated, aggregated into AADL `thread group` components, placed in address spaces (represented by AADL `processes` ) — all of these elements can be organized to into subsystems (represented by AADL `system` components) and systems.




**AADL Software System Modeling (AADL graphical view)**

A key aspect of these integration activities is using AADL `connections` to associate the output ports of one component with the input ports of another. AADL's type checking checks that connected ports are type compatible. Other AADL tools such as AGREE and BLESS can be used to specify contracts (formal specification of data constraints and input output relations) and then check that connected ports/components have compatible contracts.




**AADL Software System Modeling (AADL textual view)**

Once the software architecture is specified, HAMR can generate code for it (illustrated in later in the workflow). HAMR generates execution contexts, thread skeletons, and APIs for port communication for thread components. The developer fills in the thread skeletons to complete the component implementation.

HAMR generates code for communication between thread components using the target platform's communication mechanisms which can include both local communication as well as distributed communication (e.g., supported by middleware). Realization of inter-component communication is completely automatic. That is, in contrast to realizing threads, for communication infrastructure *all* the code is generated – there are no skeletons to fill in.

HAMR code generation may be repeated as models are updated. HAMR code generation organizes code and follows certain rules to avoid overwriting developer-written code that may be written after previous code generation steps.

**Software and Hardware Integration and Allocation of Software Components to Platform and Hardware Resources**

## AADL Analysis

Using AADL as the modeling framework for HAMR allows developers to utilize a variety of AADL-based model analysis that complement code generation and support broader system engineering activities. Some of the analyses most commonly used with HAMR are:

- Model-based hazard analysis using AADL's Error Modeling Annex and supporting analyses
- Information flow analysis that uses AADL's flow annotations
- Timing and scheduleability analyses with tools like Adventium's FASTAR that work off of AADL's timing annotations and thread properties

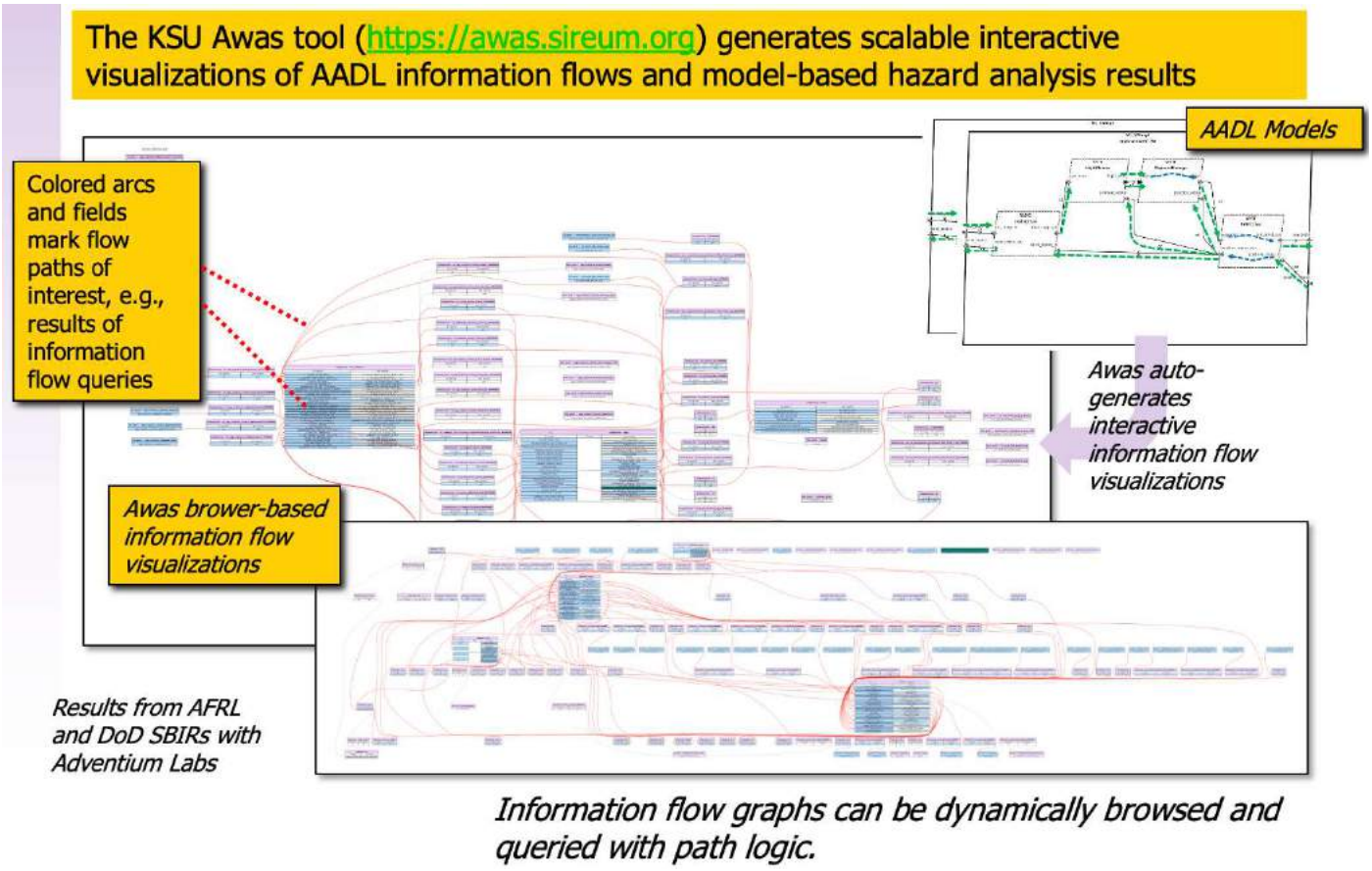
- Component behavior specification (e.g., component contracts) and verification using tools such as AGREE and BLESS

Typical workflows will likely use *some* AADL analyses before generating code (these may generate additional information, e.g., properties using scheduling analysis, that code generation may utilize), while other analyses may be used after code generation or during repeated code generation steps.

## Information Flow Analysis

Awas information flow analysis (awas.sireum.org (http://awas.sireum.org)) generates scalable interactive visualizations of AADL information flows and model-based hazard analysis results. Given an AADL model with information flow annotations, Awas will generate a Javascript/HTML5 browser-based visualization of the models information flow that you can dynamically browse and query with path logic.

Awas information flow analysis is an example of the synergistic interplay between HAMR code generation and AADL analyses – HAMR code generation for communication infrastructure ensures that inter-component information flows as visualized by Awas are actually preserved (no new flow paths are added) in the executable system (note: this does not consider various side channels).



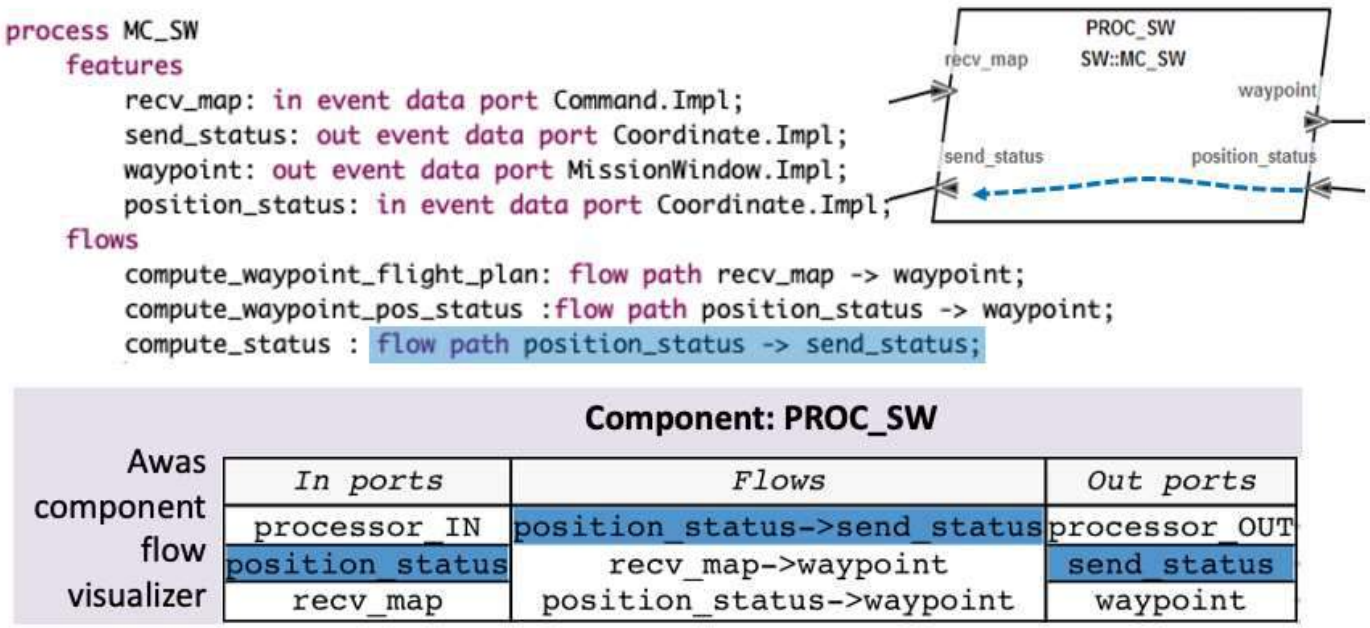

**Awas provides scalable visualizations of information flow specified in AADL modeling**

Awas works by generating an internal information flow graph derived from the declared connections between components in the AADL model (capturing *inter*-component flows) and AADL component flow annotations (capture *intra*-component flows – flows between a component's input and output ports).

In the current state of the tooling, AADL flows for intra-component flows must be written by hand (one can think of this as "modeling" the component implementation's information flow) and there is no checked correspondence between AADL flow declarations and the source code of AADL component. Adventium Labs and Kansas State have

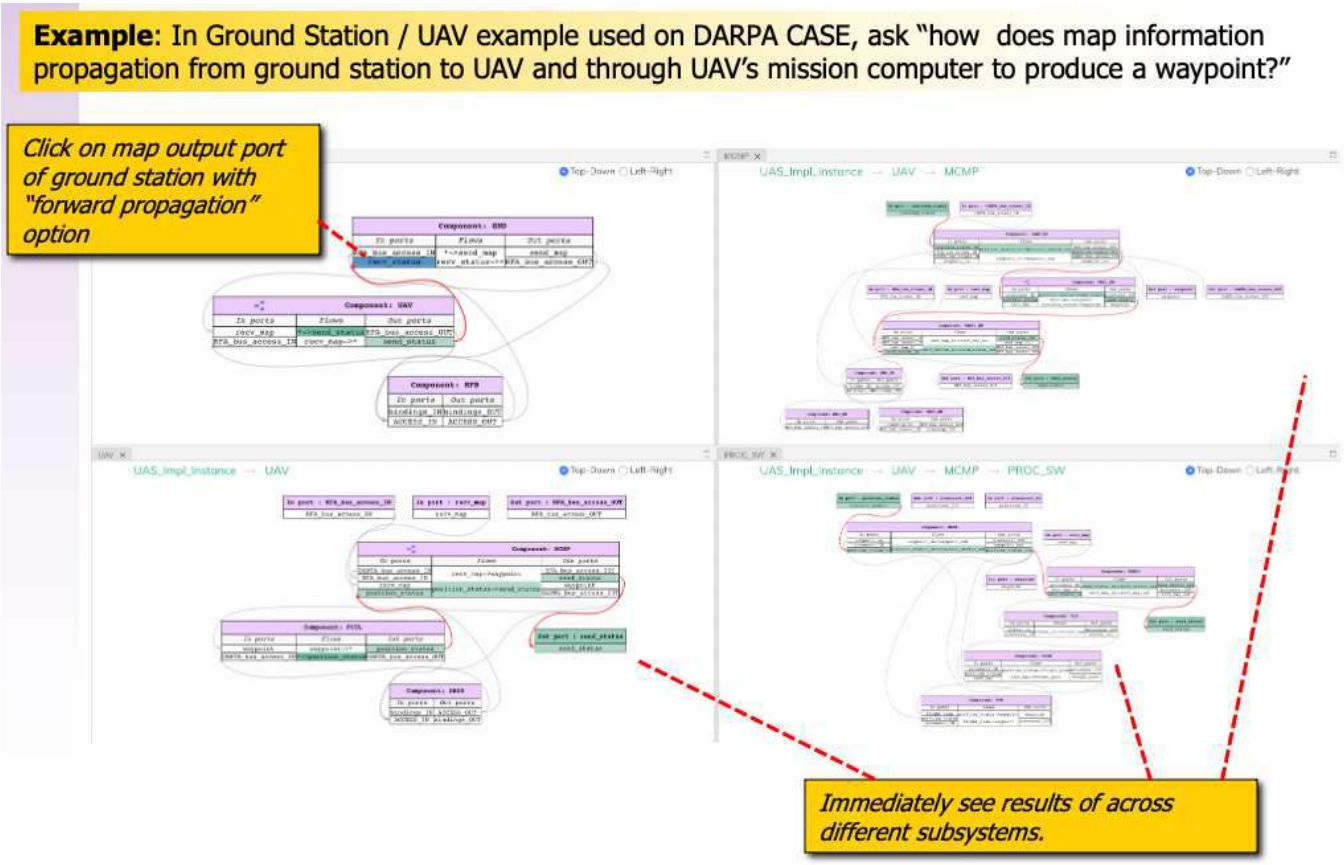
an Phase II SBIR grant from AFRL to develop (a) automatic inference of AADL flows from Slang implementations of components, as well as support for a complementary workflow in which hand-written AADL flows (serving as information flows specifications) are automatically checked against information flows found in Slang component implementations, ensuring that information flow in the Slang component implementations conforms to the AADL flow specifications.

Note that the code generation in the current tooling guarantees correspondence between the model-level *inter*-component information flows (represented as component connections) and the code-level information flow reflected in port-based inter-component communication. This correspondence is not formally proven correct at this time.




**Awas dependence graphs are derived (in part) from developer-specified AADL intra-component flow annotations**

The figure above illustrates AADL intra-component flow specifications and their corresponding representations in the Awas-generated Javascript visualizations. For example, given an AADL process component `MC_SW`, AADL flow specifications can be used to specify that information flows from the `position_status` input port to the `send_status` output port. The Awas tabular view of component inputs and outputs at the bottom of the figure provides a summary of the input/output and information flow aspects of a component. Awas query results are displayed using coloring to mark up particular set of inputs and outputs and flows that participate in a resulting information flow path.




**Awas provides interactive information flow querying (e.g., via clickable graphical representations) and visualizations of query results**

The figure above illustrates how Awas can simultaneously display information flow of multiple subsystems or multiple levels of the system hierarchy. Awas support multiple forms of queries. In the example above, in the display for the top-level components in the system architecture, the user clicks on an output port (with a *forward information flow* option being previously set), and Awas immediately calculates and visualizations the connections, ports, and components in the system that may receive information derived from the outputs of the selected port. In this example, the query corresponds to the application concern "how does map information propagation from ground station to UAV and through UAV's mission computer to produce a waypoint?".

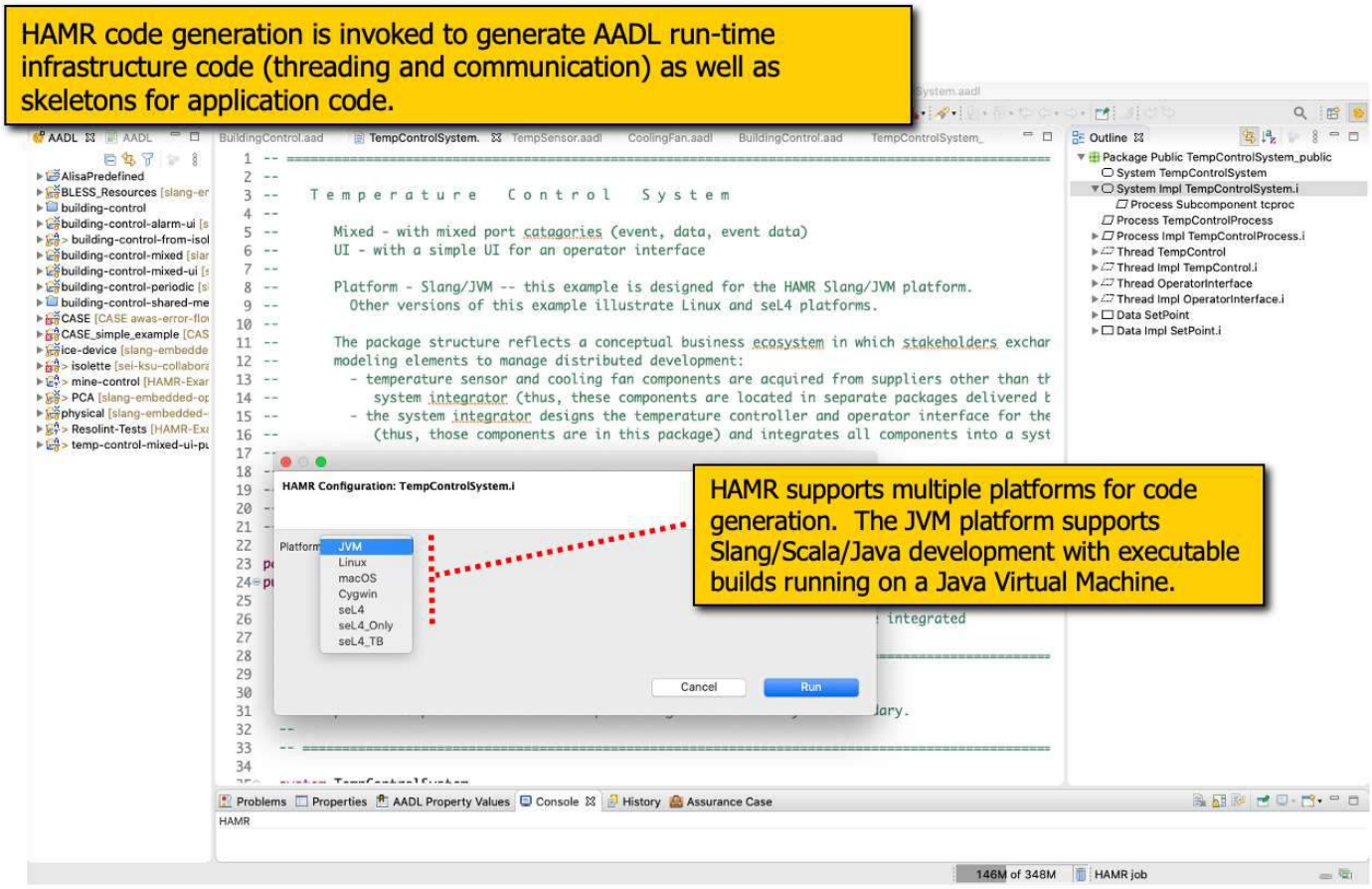
An extended walk-through of Awas browsing and querying can be found at this link (https://awas.sireum.org/doc/02-user-story/index.html) .

# Slang Workflows

This section provides an overview of HAMR capabilities that are used when developing components implementations using Slang and executing a system on the JVM platform.

## Generating Code and Code Development Environment Setup in the IntelliJ-based Sireum Integrated Verification Environment (Sireum IVE)

Once the AADL model has been developed to a level of detail sufficient to support code generation, the HAMR code generation capability can be invoked from the OSATE menu system.



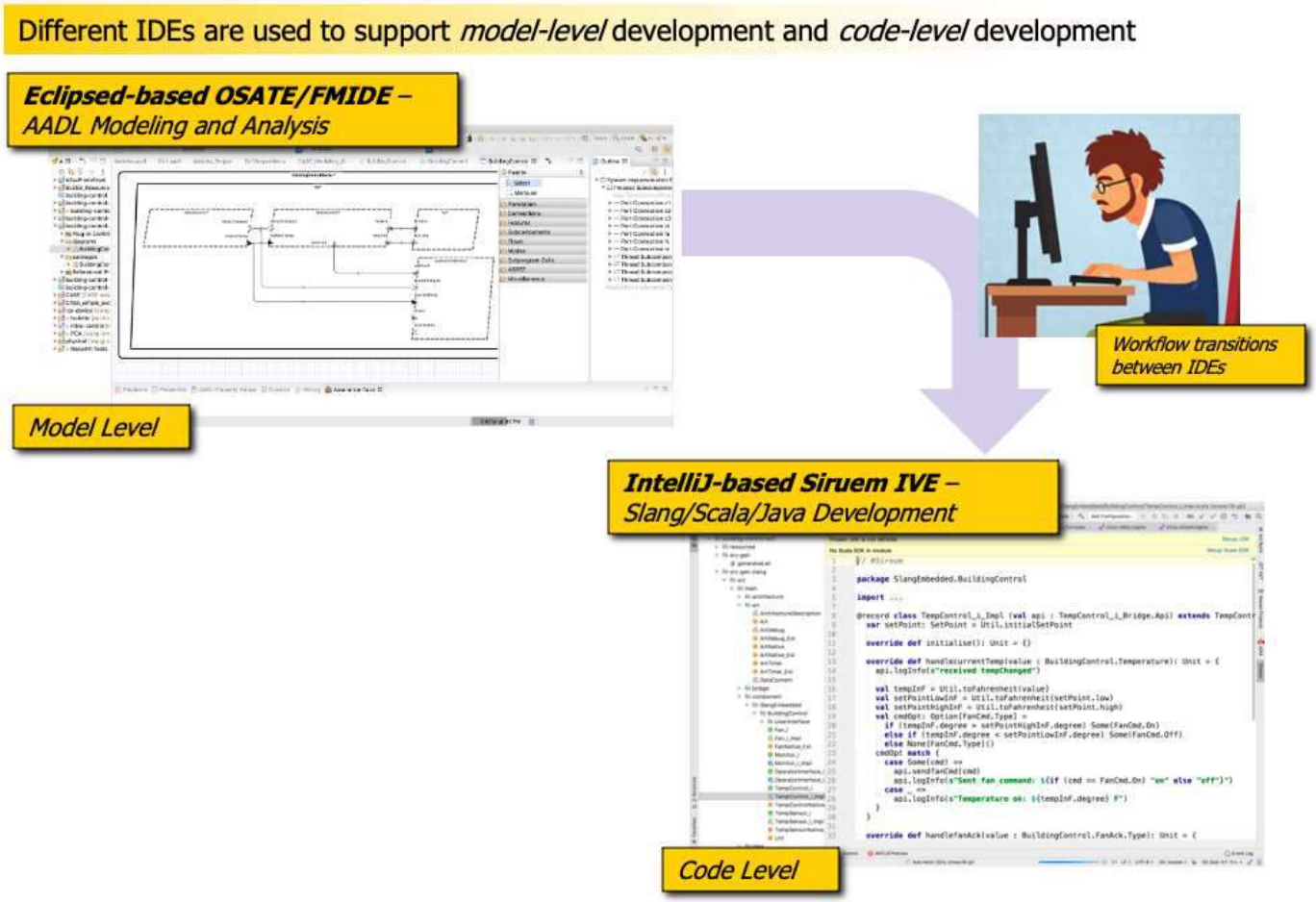

**HAMR Code Generation (Slang / JVM Platform)**

HAMR generates code for:

- **communication infrastructure code** that implements communication between AADL threads (e.g., communicating over port connections),
- **threading infrastructure code** for containers that host threads and a thread component state (one for each thread instance in the AADL model),
- **code skeletons for thread application code**,
- **build scripts** (in SBT) for configuring a build of the system and identifying needed library packages,
- **unit testing infrastructure** that aids the developer in setting up unit tests for each thread component,
- **code that supports simulation and visualization** of Slang systems.

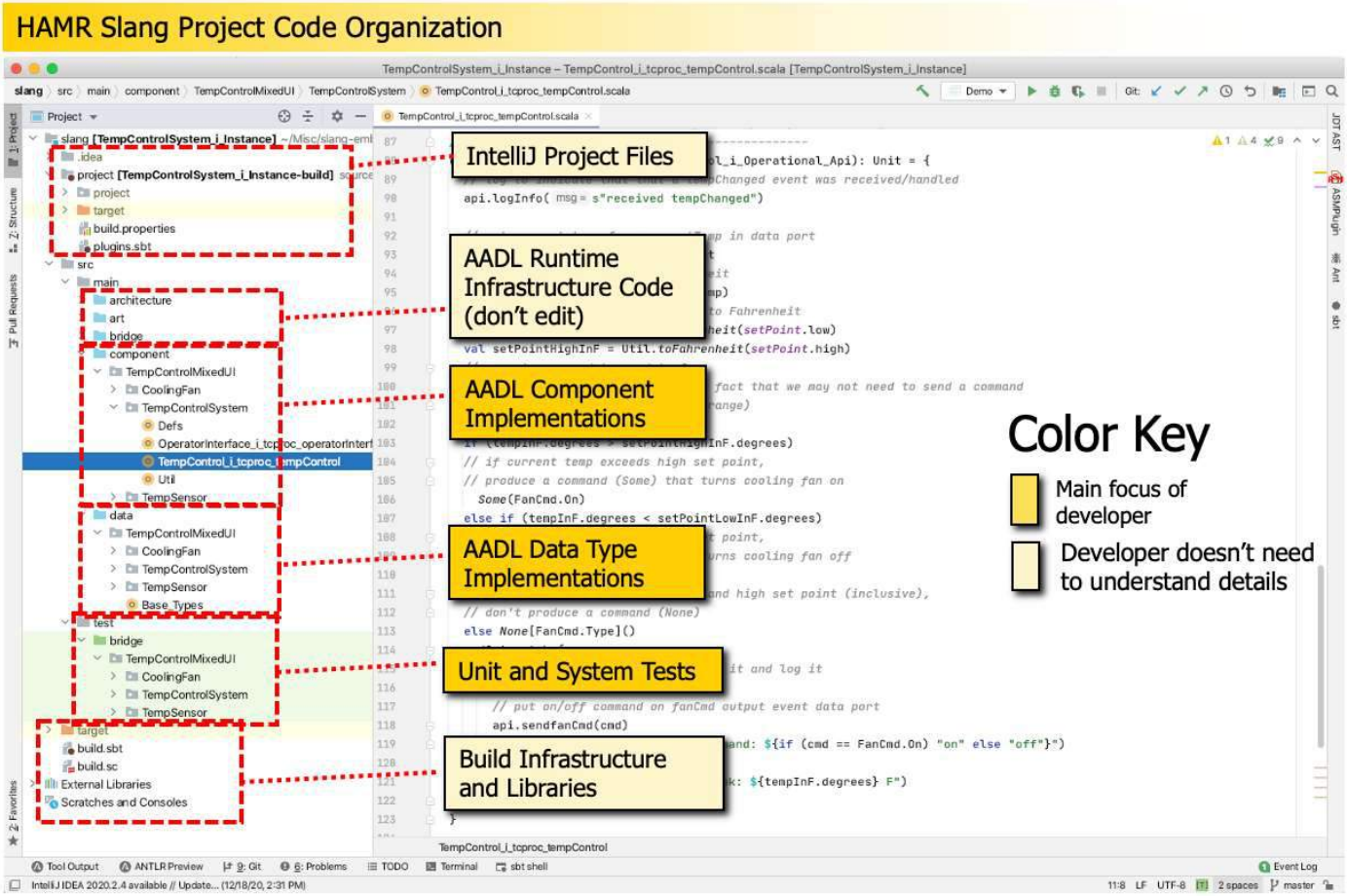
The developer will subsequently implement the system by filling in the code skeletons for the application code, configuring build scripts, adding unit and system tests, etc. The generated infrastructure code is complete and should not be modified in the development process.

For programming component logic, the workflow transitions from the Eclipse-based OSATE AADL environment (or the DARPA CASE customized version of OSATE called the FMIDE) to the IntelliJ-based Sireum Integrated Verification Environment (Sireum IVE). HAMR installation instructions cover the installation of both the FMIDE and Sireum IVE. The Sireum IVE extends IntelliJ with extensions to the Scala compiler and other facilities used to support Slang. When using HAMR's C-based workflows, there is a similar handoff from OSATE/FMIDE to C IDE's such as JetBrain's CLion.




**HAMR Workflow Transition from OSATE/FMIDE (for AADL) to Sireum IVE (for Slang/Scala/Java)**

Within the Sireum IVE, the different types of code generated by HAMR are structured in a hierarchy of files/packages. The developer's primary concern will be the AADL component implementation code, AADL data type implementations, and unit and system tests. A HAMR beginner usually doesn't need to understand the details of the other types of files.




**HAMR Slang Code Organization in the Sireum IVE (IntelliJ IDE)**

## Build Scripts

HAMR generates generates default build scripts in SBT (https://www.scala-sbt.org). SBT is similar to Maven (https://maven.apache.org) and Ant (https://ant.apache.org) – popular Java-oriented build tools. For example, SBT adopts Maven's project folder structure and supports library dependences in Apache IVY (https://ant.apache.org/ivy/) library repositories.

However, SBT builds are defined in a Scala data structures (as opposed to XML used in Maven and Ant), and this provides greater flexibility, some degree of type checking on build definitions (utilizing underlying type checking facilities of Scala), and a deeper integration with the Scala ecosystem.

The HAMR-generated SBT build script is sufficient for running the initial default behavior of the HAMR-generated system and for launching auto-generated example unit tests.

Developers extend the default build scripts as needed to specify library dependences and to tailor execution and testing workflows for the application.

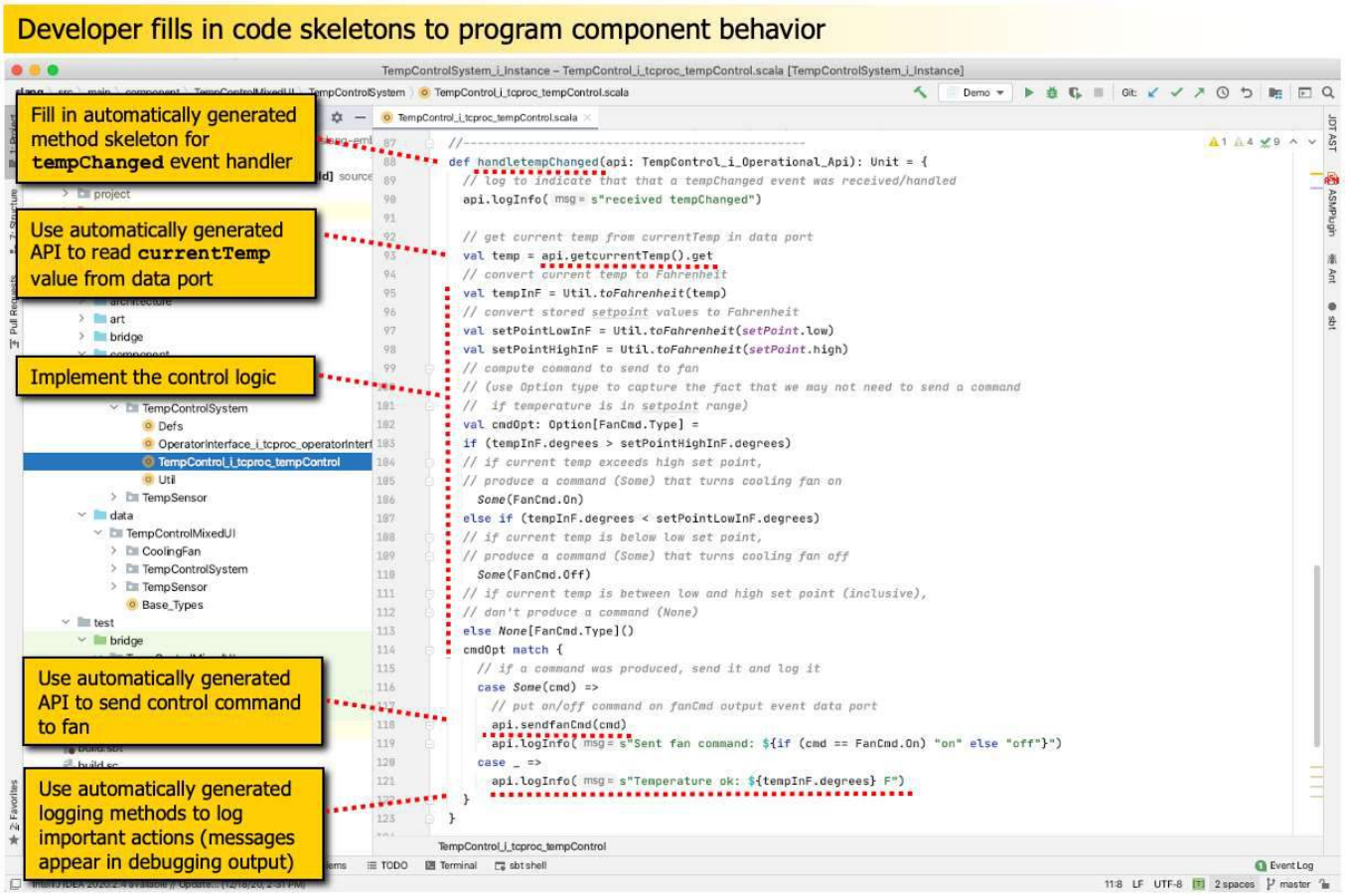
## Component Development

HAMR generates code skeletons for AADL thread components. For each thread, the structure of the thread's skeletons depends on the dispatch protocol (periodic, sporadic, etc.) selected in the AADL definition of thread.

> **Note**
>
> Add figure for component code skeletons.

The skeleton for each thread includes a default implementation consisting some simple logging commands and illustrations of how to use port APIs. The default implementations enable the generated code to be immediately executed. The default implementations can be replaced incrementally with complete implementations – all the while maintaining a system that can be executed.

When completing a thread implementation, the developer will typically (a) use the auto-generated port APIs to read data from input ports, (b) code the "business logic" that works on thread inputs and other component local variables, and (c) use the auto-generated port APIs to write data to output ports.




**HAMR Slang Complement Implementation (AADL Thread Components)**

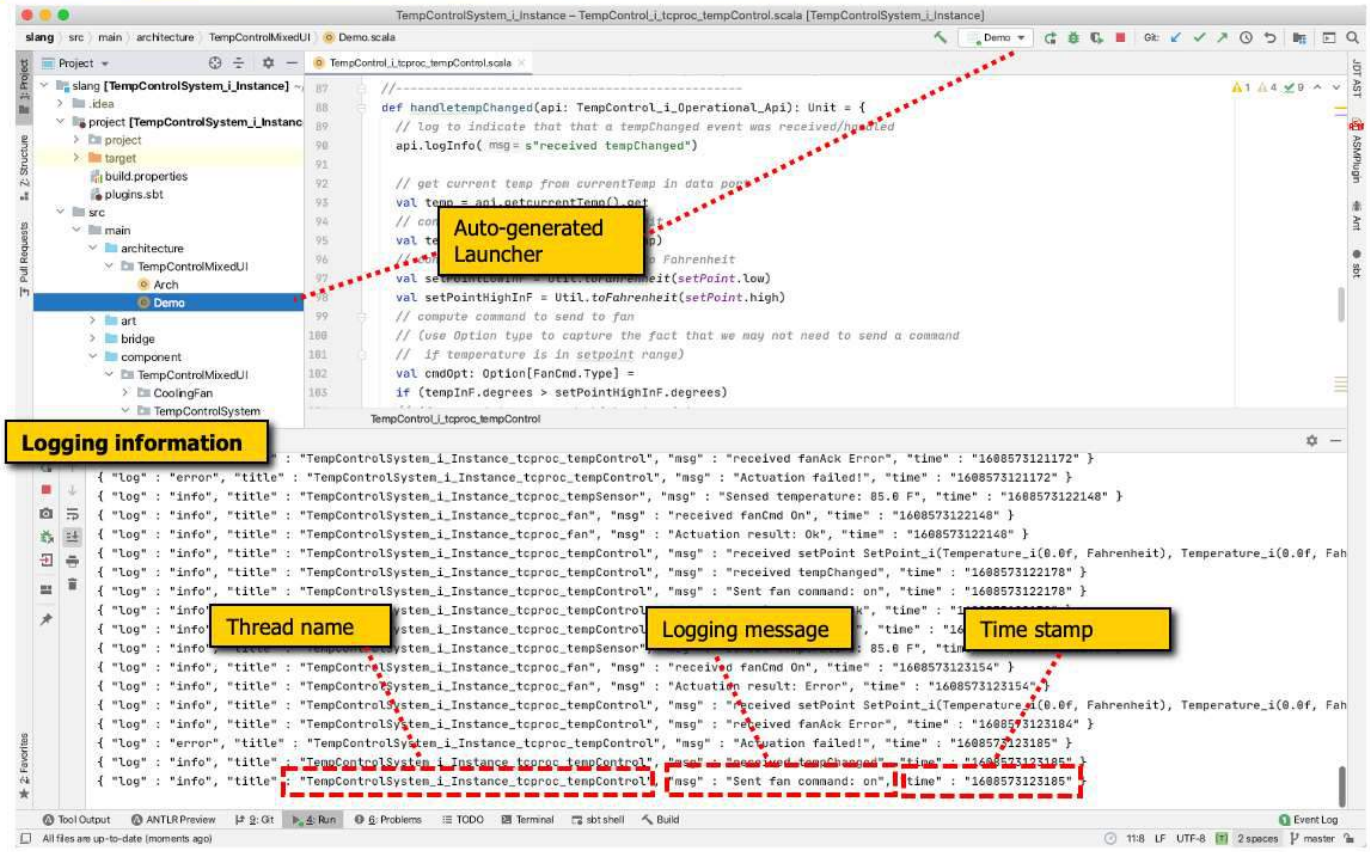
The figure aboves shows part of the component implementation (the event handler for the `tempChanged` input event port in `TempControl` component).

> **Note**
>
> Add figure for extensions and GUI components.

## System Development and Execution

HAMR build scripts integrate component implementations with auto-generated communication and threading infrastructure to produce an executable system. System execution can be started using an auto-generated launcher that can be launched within the Sireum IVE. Logging information is displayed in the console of the Sireum IVE.

The system can be easily executed in the Sireum IVE using the automatically generated launcher and build script – with logging information displayed in the IDE console window




**HAMR Slang Launcher and Basic Execution with Logging in Sireum IVE**

## Unit Testing

- IMAGE (need to draw): Explanation of component testing in terms of component inputs and outputs
- Illustration of code providing inputs and checking outputs
- Running via Scala Test

## System Testing

- System testing via controlled schedule exploration

## System Monitoring and Execution Visualization

- Inspector Framework
- MSC Visualization
- Redis logging

# C Workflows

## Generating Code and Code Development Environment Setup

- Screen shot of OSATE menu (explain multiple platform options)
- Quick overview of generated code in CLion IDE

- Setting up builds with CMAKE

## Component Development

- Coding component logic
- Reading and writing from ports
- Interfacing with Java and component GUIs

## Scheduling and System Configuration

- Configuring scheduler

## Linux System Execution

- illustration of Linux Execution

## seL4 System Execution (simulation via Qemu)

- illustration of seL4 Execution on Qemu

## seL4 System Execution (on development board)

- illustration of seL4 Execution on board

# Overview

The AADL standard describes both the syntax and semantics of AADL models. The semantic aspects of AADL can be understood as a *computational paradigim* – a constrained way of organizing processes, threads, and communication – that matches the domain properties of real-time embedded systems.

The goal of this chapter is to provide a high-level description of

- basic aspects of the AADL computational paradigm,
- how HAMR generates infrastructure code that implements the paradigm,
- how HAMR generates code skeletons and APIs to help developers write code that conforms to the paradgim,
- how developers use the HAMR generated infrastructure and APIs to develop components and systems.

These concepts will be illustrated using Slang and the associated Slang/JVM-based reference implementation of the AADL run-time (because the presentation is cleaner in Slang than in C). Details for HAMR development with Slang are found in XXXX, and details for development in C are found in XXXXX.

It's well beyond the scope of this documentation to provide a complete tutorial on AADL modeling. The following books are good resources for learning and applying AADL:

- Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language (https://www.amazon.com/Model-Based-Engineering-AADL-Introduction-Architecture/dp/0134208897/ref=sr_1_1?keywords=AADL&qid=1581775201&sr=8-1), by Peter H. Feiler and David P. Gluch. Addison-Wesley Professional, 2012.
- AADL In Practice: Become an expert in software architecture modeling and analysis (https://www.amazon.com/AADL-Practice-software-architecture-modeling-ebook/dp/B071WHRJY3/ref=sr_1_2?keywords=AADL&qid=1581775402&sr=8-2), by Julien Delange. Amazon.com Services, 2017.

The AADL OSATE (https://osate.org) IDE's integrated help system provides good documentation for how to use OSATE to build AADL models.

## Primary Information that Directs HAMR Code Generation

Even though AADL's computational paradigm is constrained to align with the needs of real-time embedded systems, it is general enough to support a range of computational styles (which one might think of AADL computational *sub-paradigms*). For example, AADL can support periodic threading with sampling data ports as found in ARINC 653 (Modular Avionics) and it can also support event-driven communication with sporadic/aperiodic threading which can be implemented by widely used message passing frameworks like ActiveMQ or the OMG's Data Distribution Service (DDS).

Developers use standard AADL property sets (properties are model annotations) to configure AADL's computational paradigm for a particular system/application. For example, AADL properties are to indicate the dispatch mode of a thread (e.g., periodic, sporadic, etc), set the size of event/message queues, set timing constraints on threading and communication, etc.

Developers use the AADL data modeling framework to specify the types of messages/data exchanged between components. HAMR supports a large portion of the data modeling approach specified in AADL's Data Modeling Annex (https://www.sae.org/standards/content/as5506/2/) (details are given in Chapter XXX). HAMR includes infrastructure to auto-generate programming language data types and wire (raw) formats from HAMR-compliant AADL data models. When using HAMR raw data formats (which are realized using Slang's bitcodec framework), libraries of encoders/decoders are also generated to aid developers between structured programming language types and corresponding raw types.

Overall, the structure of the code that HAMR generates is determined by:

- the choice/arrangement of AADL modeling elements (i.e., what "boxes and lines" are present in the AADL model and their containment and association relationships) – these determines the structure of the programming language-level threads/processes generated and the APIs that the component developer uses to communicate via ports with other threads/processes in the component context – the most important elements wrt to programming are thread components and their ports,
- AADL properties attached to the modeling elements – these determine specific implementation strategies and resource allocation for threads/processes, communication, message buffers, etc.,
- the data types attached to AADL ports – these determine programming language types that appear on code-level APIs for inter-component communication, and libraries that are generated to support encoding/decoding for raw types, and
- the selection of language/platform for the code generation – these determine the programming language for the generated code, the implementation strategy for port-based inter-component communication, the implementation strategy for threading, etc.

Regardless of the language/platform selected, HAMR code generation aims to produce code that conforms to the AADL computational paradigm as configured by the first three items above. Even though code is generated in different languages and may be developed using different IDEs, HAMR aims to structure generated code so that APIs and workflows are similar regardless of the specific programming language being used for component development.

The most important AADL concepts for HAMR users to understand are *threading/tasking* and *communication* as realized via AADL thread components and port-based communication. This chapter will focus on those concepts, and the reader will need to refer to other sources for description of other AADL modeling concepts.

## Important Restrictions and Caveats

**Note**

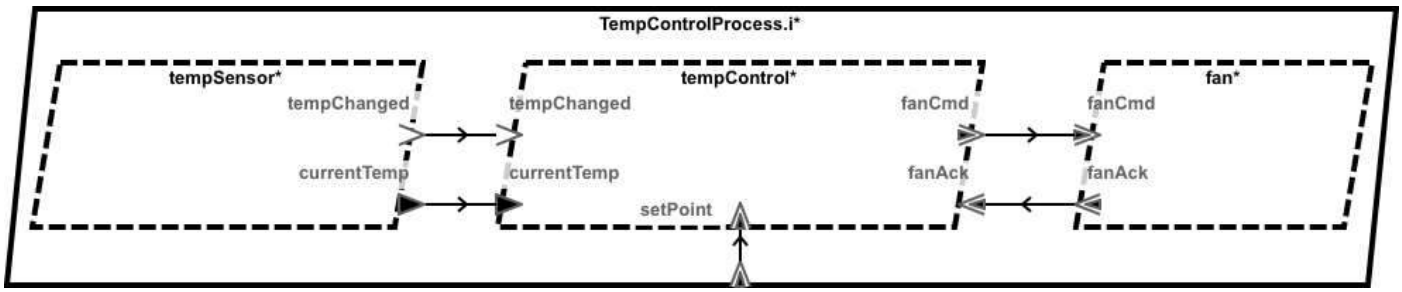
ToDo: finish this subsection

- Port-based communication only - no subprograms, no data components
- Limitations on scheduling
- Semantics of AADL in the standard are underspecified. The contents of this chapter present our interpretation. Standard currently under revision.
- Resolint used to check that HAMR's AADL inputs conform to constraints.

# Example

## Example System Functionality

This section presents a minimalistic example designed to illustrate the basic aspects of the AADL computational paradigm. The example doesn't necessarily reflect the way that one would design a system in practice. Rather, the features are chosen to provide coverage of both periodic and sporadic components and both data and event ports.




**AADL Temperature Control Example (graphical view)**

AADL Temperature Control Example (graphical view) uses the AADL OSATE graphical view to show the thread and process (the top level system component is omitted) for a simple temperature controller that maintains a temperature according to setpoints (i.e., high and low temperature values). The `tempSensor` thread measures the current temperature and transmits the reading on its `currentTemp` data port. It also sends an event on its `tempChanged` port if it detects the temperature has changed since the last reading. Note: `tempChanged` and `currentTemp` could alternatively be combined into single `event data` port.

When the `tempControl` thread receives a `tempChanged` or `setPoint` event it compares the value on `currentTemp` with the high and low temperatures from aadlPort{setPoint} and sends `FanCmd.On` or `FanCmd.Off` to the `fan` thread via its `fanCmd` event port. The `fan` acknowledges whether it was able to fulfill the command by sending `FanAck.Ok` or `FanAck.Error` on its `fanAck` event port.

## Important AADL Specification Concepts

The typical focus of the initial AADL modeling effort is to lay out the application *threads/tasks* of the system (e.g., the `TempSensor`, `TempControl` and `Fan` threads in the example above) and port-based *interactions* between those threads. Many things happen beyond that, but that is often the focus of the developer's activities – especially as they are thinking about what needs to be computed and communicated. Beyond these basic steps, AADL `thread groups` can aggregate similar threads as a modeling abstraction. AADL `process` components then indicate the memory spaces within which the threads will work. AADL `systems` add further architectural hierarchy by representing subsystems and the top-level system. AADL resource layer which includes notions such a

`processor`, `bus`, etc. can be used to model platform resources, and AADL *bindings* are used to allocate software components to resources. Along the way, AADL *properties* are added to model elements to configure them in various ways.

AADL supports both a graphical and textual views of models. The graphical view of the temperature controller example is given in the previous AADL Temperature Control Example (graphical view). The graphical view is typically used for high-level understanding of models or for rapidly building up an architecture by dragging/dropping for a pallete of modeling elements. A lot the detailed engineering work happens in the textual view because it is more amenable to attaching and viewing various forms of model properties, semantic information, and analysis directives.

## AADL Declarative Models

The listing below shows a portion of the AADL textual view of what is called the *declarative model* for the building control example (The full model is available in Appendix XXXX).

```
process TempControlProcess  -- component type for temperature controller subsystem (modele
features
   setPoint : in event data port SetPoint; -- the subsystem contains one input (the high a
end TempControlProcess;

process implementation TempControlProcess.i -- the component implementation has subcompone
subcomponents
   tempSensor : thread TempSensor.i;
   tempControl : thread TempControl.i;
   fan : thread Fan.i;
connections
   sp : port setPoint -> tempControl.setPoint;
   ct : port tempSensor.currentTemp -> tempControl.currentTemp;
   tc : port tempSensor.tempChanged -> tempControl.tempChanged;
   fc : port tempControl.fanCmd -> fan.fanCmd;
   fa : port fan.fanAck -> tempControl.fanAck;
end TempControlProcess.i;

thread TempSensor
features
   currentTemp : out data port Temperature;
   tempChanged : out event port;
properties
   Dispatch_Protocol => Periodic;
end TempSensor;

thread implementation TempSensor.i
-- Nothing further to model in AADL for the TempSensor implementation.
-- This is a leaf node in the architecture model -- the implementation is realized
-- directly as source code.
end TempSensor.i

thread Fan
features
   fanCmd : in event data port FanCmd;
   fanAck : out event data port FanAck;
properties
   Dispatch_Protocol => Sporadic;
end Fan;

thread implementation Fan.i
-- Nothing further to model in AADL for the Fan implementation.
-- This is a leaf node in the architecture model -- the implementation is realized
-- directly as source code.
end Fan.i

thread TempControl
features
   currentTemp : in data port Temperature;
   tempChanged : in event port;
   fanAck      : in event data port FanAck;
   setPoint    : in event data port SetPoint;
```

```
   fanCmd      : out event data port FanCmd;
properties
   Dispatch_Protocol => Sporadic;
end TempControl;

thread implementation TempControl.i
-- Nothing further to model in AADL for the TempControl implementation.
-- This is a leaf node in the architecture model -- the implementation is realized
-- directly as source code.
end TempControl.i
```

A declarative model describes the software and hardware components of a system along with their interactions and organization (Listing XXXX only addresses software aspects since those are sufficient for the discussion of the AADL computational paradigm). AADL *instance models* are automatically derived (e.g., by the OSATE tool) from a declarative model by instantiating the declarative model for a particular instance (i.e., configuration) of a system. The instance model concept is necessary because each AADL component has a type declaration and one or more implementation declarations, and the instance model reflects a choice of implementation for each component type. Instance models also "flatten" or "peel off" AADL constructs that are mainly used to organize and indicate containment such as sub-systems, processes, thread groups and feature groups – leaving only primary elements such as threads and port-connections. Instance models are discussed more detail below. For the most part, for simple AADL models and applications of HAMR, it is safe for developers to ignore the details of an instance model and think in terms of the declarative model. The only time an instance model comes is to play is when preparing to generate code – the developer selects a particular system implementation from which an instance model is generated. Even then, the developer usually doesn't need to understand the details of the instance model – only that a particular instance/configuration is selected for which code will be generated.

## Component Types

The external interfaces of components (called *component features*) are specified via *component type* declarations. For example, the component type declaration `TempControl` has five ports that can be used when interacting with other components. The direction information flows through the port is specified using the keywords `in` or `out` (Note: AADL also has `in out` (i.e., bi-directional) ports but those are not supported by HAMR because they complicate system semantics and information flow reasoning). Data ports (indicated by the `data` keyword) are used to communicate state data between components without queueing (old values on the receiver side are overwritten with newer values). Event data ports (indicated by the `event data` keywords) asynchronously transmit messages between components, where messages are buffered/queued on the receiving component. Event ports (indicated by `event` keyword) allow a sending component to signal a receiving component in a message without a payload.

## Component Implementations

AADL's `implemention` constructs model specific implementations of a component type. For each component category (e.g., `system`, `process`, `thread`, etc.), the AADL language includes a corresponding implementation construct (e.g., `system implementation`, `process implementation`, etc.) Sometimes a component implementation models the internal structure of a component as a collection of contained connected subcomponents (e.g., the component implementatation `TempControlProcess.i.` implements the component type `TempControlProcess` and contains the architectural containment hierarchy of the process). In other cases, component implementations don't specify the internal structure of the component because the component is a *leaf node* of the architecture, and the component will be directly implemented in source code or hardware. In such

cases, the body of the component implementation may be left empty or other AADL properties may be used to indicate the filename of the code that implements the component (e.g., the implementations of the `TempSensor`, `Fan`, and `TempControl` components). A component type may have multiple corresponding implementations, but this capability is often not used in simpler models. For leaf nodes, different implementations might be used to model the fact that different algorithms are available to choose from for a particular thread. For components with containment, different implementations might be used to indicate selections of different algorithms for its subcomponents or to model different possible architectures for subsystems that, e.g., provide a both straightforward implementations as well as a different topology with redundant components to achieve fault-tolerance.

In implementations with containment, interaction between the subcomponents is declared via named port-to-port *connections*. For example, in `TempControlProcess.i` there are three thread subcomponents (`tempSensor`, `tempControl`, and `fan`). Named connections represent the abstract communication pathways between the components (e.g., in the `TempControlProcess` implementation, the connection named `ct` represents a communication link between the `currentTemp` output port of `tempSensor` and the `currentTemp` input port of `tempControl`.

The `dispatch_protocol` settings of `Periodic` and `Sporadic` are examples of AADL *properties* that can be added to the model to support model analyses and to direct code generation or other aspects of the development tool chain. In these particular examples, `Periodic` and `Sporadic` inform AADL schedulability analysis that the underlying component threads will be dispatched by either periodic timing events (periodic) or by the arrival of events on event data ports (sporadic). They direct code generation tools to insert different types of hooks from the scheduler and communication infrastructure to the application code realizing the component logic. This use of properties along with coordination between analysis and code generation is key to enabling model analysis and to ensuring that analysis results apply to the generated code.

### AADL Instance Models

**Note**

ToDo: Finish this section.

In addition to the declarative model format, AADL also has an instance model format that is generated from declarative model. An AADL instance model accomplishes several things.

- **To Do:** explain multiple component implementations and how an instance model picks the specific implementation.
- **To Do:** explain how an instance model tunnels through the architecture hierarchy to focus on threads. When generating code from AADL models, the thread components are the model elements that dictate the most of the structure of the generated code.

## AADL Threading

This section addresses the structure of the HAMR-generated code for an AADL thread component. The code structure is influenced by two AADL concepts:

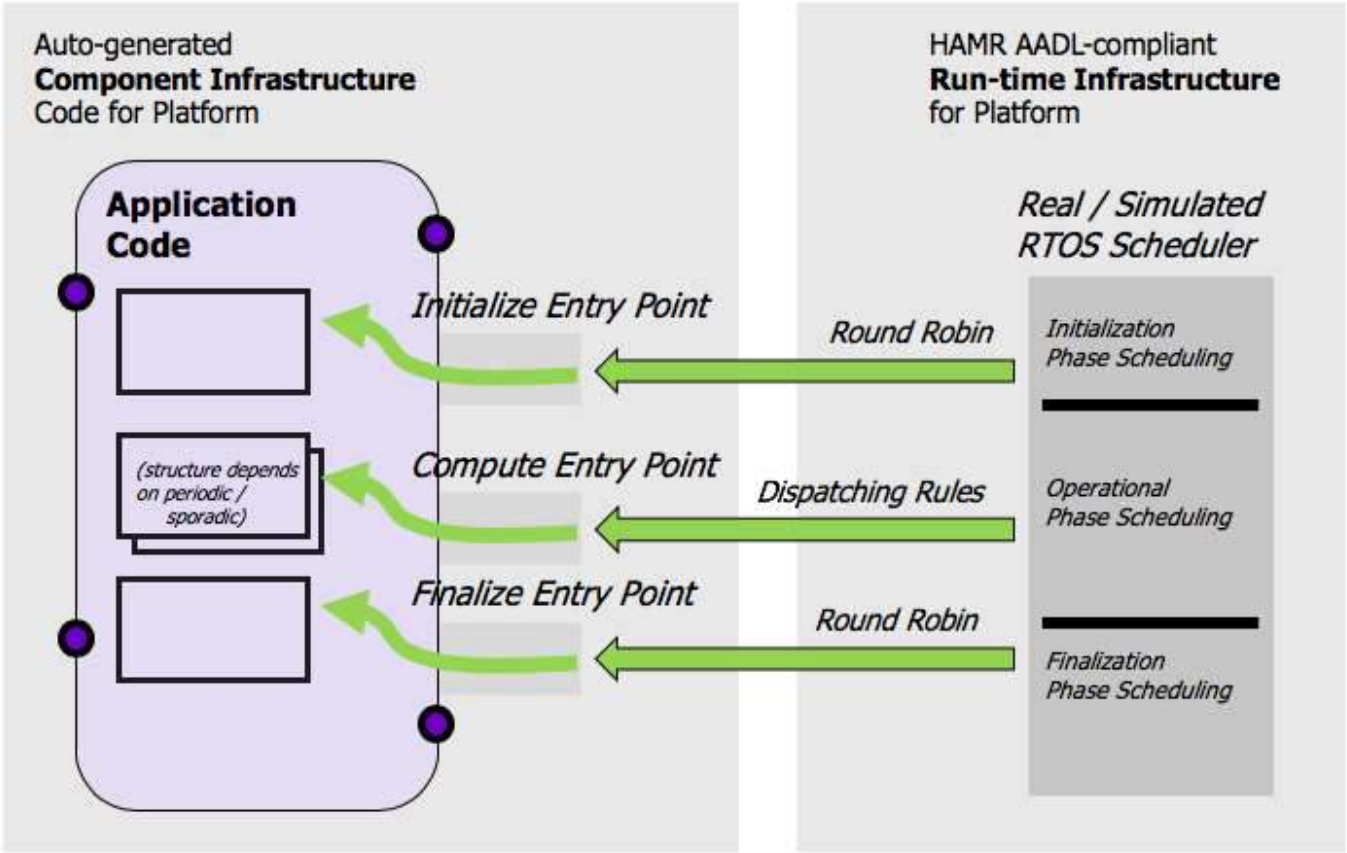
- the AADL concept of *entry point* – i.e., the different application functions/methods that the underlying run-time infrastructure (e.g., RTOS scheduler) calls to run the application code for the thread, and
- the AADL concept of thread *dispatch protocol* – the rules (protocol) that determine when the thread application code is executed (e.g., whether a thread is classified as *periodic* or *sporadic*).

In general, the code that HAMR generates to support threading does various things behind the scenes (i.e., the actions are purposely hidden from the developer so that they don't accidently change the associated code, and application code doesn't need to understand the details) including receiving and propagating port data and taking steps to ensure that application code can be structured in a way that facilitates analysis and verification. Depending on the programming language targetted, HAMR uses different language mechanisms to isolate the "behind the scenes" threading infrastructure code from the developer-written application code so that the developer can concentrate on the application logic of a component.

Since it targets embedded systems, the AADL standard indicates that threading should be supported by an AADL-compliant run-time environment that sets up threading in a manner that ensures computational paradigm. This means that developers will not use the threading mechanisms of the programming language directly, i.e., C programmers will not directly create Posix threads, Scala programmers will not use Scala (JVM) threads, etc. The AADL run-time may use those mechanisms in the infrastructure code to achieve the computational paradigm, but those details are not exposed to the application code and developers typically do not need to be concerned about them. All the developer needs to understand is the AADL thread abstraction, and how to configure/constrain the thread abstraction and scheduling using AADL thread properties (e.g., priority, deadline, etc).

## Thread Entry Point Concepts

The interfacing between thread application code and AADL run-time is defined in the standard using the concept of *entry point* – which is often used in real-time tasking. In other frameworks and OSes, the application code makes the scheduler aware of the entrypoints by registering certain methods with the scheduler. With HAMR, entry point skeletons are auto-generated for the developer using certain naming conventions, and the scheduler invokes entry points with those specific names.




**HAMR AADL-compliant Thread Entry Points**

HAMR AADL-compliant Thread Entry Points illustrates some of the primary concepts of HAMR thread entry points. First, it's important to understand that the AADL standard (see Clause 5.4.1 (21)) – following common practice in avionics and other safety-critical systems – recognizes distinct phases of execution: an *initialization phase* and an *operational phase* (see the right side of the figure). In the initialization phase, each thread's infrastructure is initialized and the Initialize Entry Point of each thread is called to execute developer-supplied code that initializes the application state of the component. After all thread initialization is complete, the system shifts to the Operational Phase. The standard does not specify a particular scheduling policy/order for initialization, each thread's Initialize Entry Point is called exactly once, typically in a round-robin fashion. During the operational phase, the Compute Entry Point of each thread (which implements the application logic of a thread) is called according to a developer-configured scheduling policy, set or derived from various AADL properties or other means (e.g., a static schedule provided as part of the platform infrastructure configuration). In the current implementation of HAMR, after the operational phase is complete, the Finalize Entry Point for each thread is called.

Thus, from the developer's point of view, the business logic of the thread is programmed by implementing the following entry points (HAMR code generation provides a skeleton for each of these).

- Initialize Entry Point - used to initialize any thread-local variables, initialize drivers or other resources that the thread manages, and used to set the output ports with initial values. The **values of output data ports must be set**, whereas sending messages on output event or event data ports is optional. **Input ports must not be read in the Initialize Entry Point**. These read and write constraints are currently not enforced by static code analysis in the HAMR supported languages, although it is likely that they will be checked in the future for Slang. The time at which the output port values are propagated to consumers is not specified in the standard – only that propagation is guaranteed before the operational phase starts.
- Compute Entry Point - used to implement the primary business logic of the component. Port APIs, described in the following section are used to exchange information with other components to which the thread is connected. In contrast to the other entry points, the structure of a thread's Compute Entry Point differs depending on the dispatch mode (periodic or sporadic) of the thread as explained below.
- Finalize Entry Point - used to for any logic needed to achieve a graceful shutdown of the thread. Currently, HAMR will not propagate any outgoing port communication initiated during the finalize entry point.

The AADL standard specifies three additional entry points that HAMR does not currently support (see Section 5.4.1 of the standard): A Recover Entry Point allows the developer to code recovery steps to be initiated on certain types of failures, and Activate and Deactivate Entry Points are used to support mode switching.

For each HAMR-supported programming language, HAMR will generate method skeletons for the entry points for each thread component. For example, for the Slang code generation of the `TempControl` component, the HAMR-generated code will include the follow in a `TempControl_i.scala`.

```
@msig trait TempControl_i { // Slang interface collecting AADL standard thread entry point

  ...

  def initialise(): Unit = {}   // signature / skeleton for Initialize Entry Point

  // Compute Entry Point not shown here -- code generation is handled in a different way t

  def finalise(): Unit = {}     // signature / skeleton for Initialize Entry Point

  ...
}
```

These are skeletons for the Initialize Entry Point and Finalize Entry Point that the developer will override in a companion file `TempControl_i_Impl.scala` to provide the implementation of these methods (along with the Compute Entry Point implementation) (see the following section).

## Initialize Entry Point

```
// TempSensor thread completed Initialize Entry Point
override def initialise(): Unit = {
  val temp = TempSensorNative.currentTempGet()
  api.setcurrentTemp(temp)
}
```

```
// TempControl thread completed Initialize Entry Point
override def initialise(): Unit = {
 setPoint = SetPoint(
    Temperature(55f, TempUnit.Fahrenheit),
    Temperature(100f, TempUnit.Fahrenheit))
}
```

## Compute Entry Point – Thread Dispatch Modes

The manner in which a thread's Compute Entry Point is invoked by the AADL run time as well as the HAMR-generated structure of the entry point itself is determined by the thread's *dispatch protocol*.

The developer sets the dispatch protocol for a thread by attaching the AADL property `Dispatch_Protocol` in the thread component type (note: according to the AADL standard, the dispatch protocol may also be set in a thread component implementation, but HAMR requires it to be in a component type because it determines the semantics of the component and the allowed structure of component contracts – thus it needs to be exposed in the "public interface" of the component as represented by the type). For example, the following excerpts from the AADL model of the temperature control example illustrate that the `TempSensor` component is declared to be `Periodic` whereas the `TempControl` thread is sporadic.

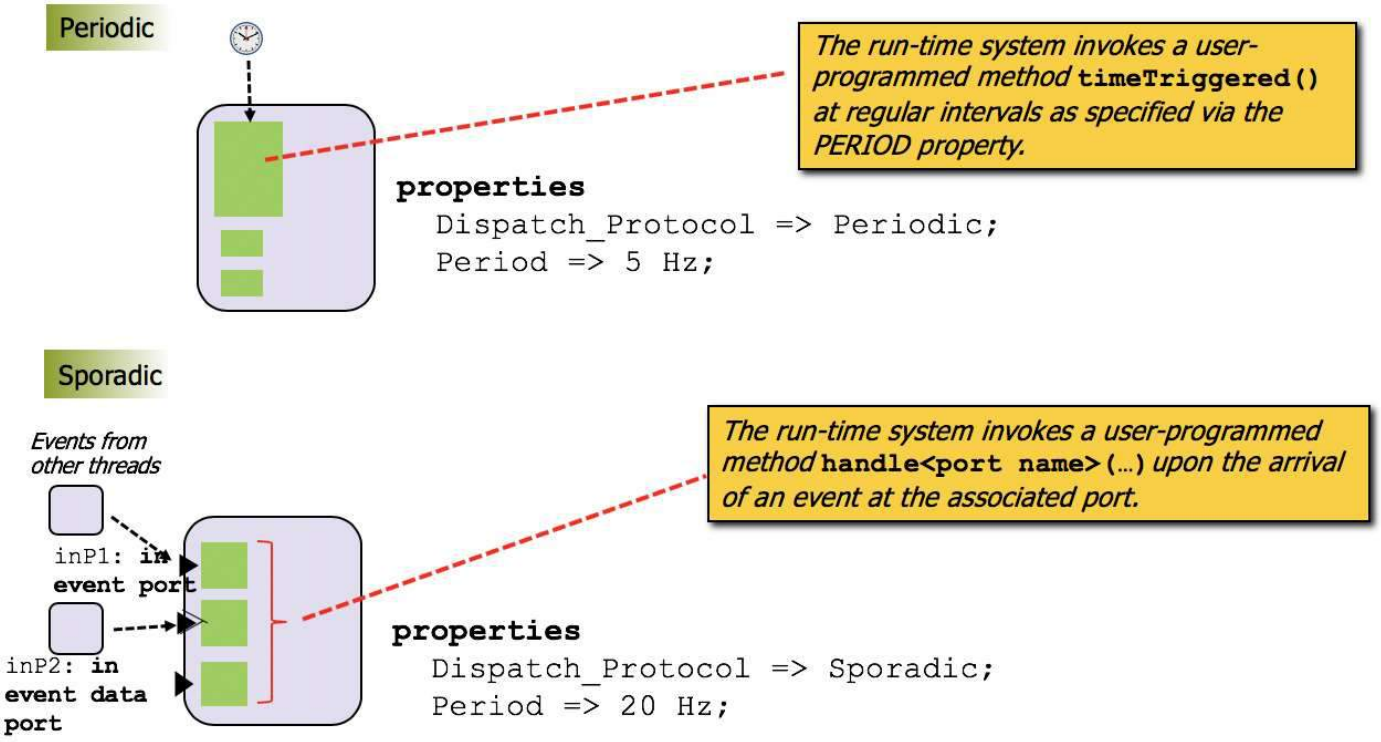
```
thread TempSensor
features
   currentTemp : out data port Temperature;
   tempChanged : out event port;
properties
   Dispatch_Protocol => Periodic;
end TempSensor;

thread TempControl
features
   currentTemp : in data port Temperature;
   tempChanged : in event port;
   fanAck      : in event data port FanAck;
   setPoint    : in event data port SetPoint;
   fanCmd      : out event data port FanCmd;
properties
   Dispatch_Protocol => Sporadic;
end TempControl;
```

The AADL standard specifies five different dispatch protocols, but only the following two are supported by HAMR.

- Periodic Dispatch - threads are released for scheduling (in AADL terms, *dispatched*) at a regular interval indicated by the `Period` property.
- Sporadic Dispatch - threads are released for scheduling when messages arrive on the thread's input event or input event data ports. AADL provides additional properties including notions of port priorities that allow finer grain control over the conditions for triggering a thread dispatch. A minimum time between dispatches can also be specified to ensure the thread can function properly in situations where a failing sender component or an adversary might flood the thread with incoming messages.

Other AADL dispatch protocols not currently supported by HAMR are Aperiodic Dispatch (similar to Sporadic but does not support the notion of a minimum interval between dispatching), Timed Dispatch (like Aperiodic, but an additional time out property can be specified that is reached will cause the thread to be dispatched), and Hybrid Dispatch (like Periodic and Aperiodic combined – the thread is dispatched both periodically and also upon the arrival of messages on event or event data ports). See Section 5.4.2 of the AADL standard for details.




**Code Structure for Periodic and Sporadic Thread Dispatch Modes**

The following subsections discuss the structure of the HAMR-generated Compute Entry Points for HAMR supported protocols along with AADL properties that are used to further configure dispatching conditions or to specify thread properties that inform schedulability and resource constraint analysis. Figure Code Structure for Periodic and Sporadic Thread Dispatch Modes summarizes concepts code structure that are discussed in greater detail below.

## Periodic Dispatch Application Code Structure

AADL threads with periodic dispatch mode correspond directly to the conventional notion of periodic thread from the real-time systems domain. The thread is *time-triggered* – it is dispatched (released for scheduling) at a regular rate, determined by the thread's period which is provided as an AADL property. To program the application logic of a periodic component, the developer implements the `timeTriggered` method whose skeleton is auto-generated by HAMR (the method has slightly different names depending on the target programming language). The `timeTriggered` method is the developer's view of the Compute Entry Point when a thread as a `Periodic` dispatch protocol.

For example, for the periodic `TempSensor` thread, HAMR generates the following Slang method skeletons for entry points.

```
// HAMR-generated entry point skeletons in TempSensor_i.scala
@msig trait TempSensor_i {

   // reference to APIs to support port communication
   def api : TempSensor_i_Bridge.Api

   // skeleton for Initialize Entry Point
   def initialise(): Unit = {}

   // skeleton for developer's view of the compute entry point (following the pattern for
   def timeTriggered() : Unit = {}

   // skeleton for the Finalize Entry Point
   def finalise(): Unit = {}
}
```

Then, the Slang application logic of the `TempSensor` is coded filling in the auto-generated skeleton for the `timeTriggered` method as follows.

```
 // TempSensor application logic (excerpts) in TempSensor_i_Impl.scala

 override def timeTriggered(): Unit = {  // this signature/skeleton is auto-generated
 // ==== developer programs application code below ===

  // get current temperature value from physical sensor using driver (not modeled in AADL)
  val temp = TempSensorNative.currentTempGet()
  // set the value of the `currentTemp` in AADL output data port using auto-generated API
  api.setcurrentTemp(temp)
  // send event on `tempChanged` on AADL output event port using auto-generated API
  api.sendtempChanged()
  // use internal library to convert raw temp to appropriate display format
  val degree = Util.toFahrenheit(temp).degree
  // log value using auto-generated logging infrastructure
  api.logInfo(s"Sensed temperature: $degree F")
}
```

While the `timeTriggered` method forms the *developer's view* of the Compute Entry Point, behind the scenes, the HAMR-generated infrastructure for a periodic component includes a `compute` method that represents the *infrastructure's view* of the Compute Entry Point. The Slang version of this method is shown below.

```
// HAMR-generated infrastructure method -- hidden from developer (in TempSensor_i_Bridge.s
def compute(): Unit = {
   // AADL run time `receive input` makes infrastructure ports values visible to applicati
   // ...this implements AADL's notion of "port freezing"
   Art.receiveInput(eventInPortIds, dataInPortIds)
   // Compute Entry Point application code for periodic component is executed
   component.timeTriggered()
   // AADL run time `sendOutput` releases values set by application code port APIs to infr
   // to consuming components
   Art.sendOutput(eventOutPortIds, dataOutPortIds)
 }
```

The delimiting of the `timeTriggered` application code by the AADL run time routines `receive input` and `send output` is an important aspect of the AADL standard's computational paradigm (the rationale for this is discussed in a following subsection after structure of sporadic threads is presented). Placing these calls in the auto-generated `compute` method in the infrastructure (separate from application code) guarantees that the computational paradigm is always followed.

## Sporadic Dispatch Application Code Structure

An AADL thread with sporadic dispatch mode is dispatched upon the arrival of messages on its input event or event data ports. In other words, it's execution is *event-triggered* instead of *time-triggered*. To tailor the application code structure to the event-driven character of the thread, HAMR generates a message handler method skeleton for each input event and event data port. To program the application logic of a periodic component, the developer completes the implementation of these method handlers. Thus, in contrast to a periodic component where the `timeTriggered` method is the developer's view of the Compute Entry Point, for a sporadic component the message handler methods form the developer's view of the Compute Entry Point.

For example, for the sporadic `TempControl` thread, HAMR generates the following Slang method skeletons for entry points.

```
// TempControl_i.scala - HAMR-generated method signatures and default implementations for
//   including event handlers that support the implementation of the Compute Entry Point

@msig trait TempControl_i {

   // reference to APIs to support port communication
   def api : TempControl_i_Bridge.Api

   // skeleton for Initialize Entry Point
   def initialise(): Unit = {}

   // skeletons for developer's view of the compute entry point (following the pattern for
   //   ...an message handler is generated for each input event and event data port

   // message handler for the `fanAck` input event data port -- `value` is the message pay
   def handlefanAck(value : BuildingControl.FanAck.Type): Unit = {
      // auto-generated default implementation simply logs the incoming message payload
      api.logInfo(s"received ${value}")
      api.logInfo("default fanAck implementation")
   }

   // message handler for the `setPoint` input event data port -- `value` is the message p
   def handlesetPoint(value : BuildingControl.SetPoint): Unit = {
      // auto-generated default implementation simply logs the incoming message payload
      api.logInfo(s"received ${value}")
      api.logInfo("default setPoint implementation")
   }

   // event handler for the `tempChanged` input event port
   //  -- there is no parameter because an AADL event does not have a payload
   def handletempChanged(): Unit = {
      // auto-generated default implementation simply logs a message that the event has be
      api.logInfo("received tempChanged")
      api.logInfo("default tempChanged implementation")
   }

   // skeleton for Finalize Entry Point
   def finalise(): Unit = {}
}
```

Then, in addition to completing the code for the Initialize and Finalize Entry Points, the primary Slang application logic of the `TempControl` is coded by filling in the auto-generated skeleton for the message handler methods as follows (only excerpts are shown).

```
 // TempControl application logic (excerpts) in TempControl_i_Impl.scala

// event handler for the `tempChanged` input event port
//  -- there is no parameter because an AADL event does not have a payload
override def handletempChanged(): Unit = {// this signature/skeleton is auto-generated

  // ==== developer programs application code below ===
  api.logInfo(s"received tempChanged")  // retain auto-generated call to logging

  // use port APIs to read the value of the `currentTemp` input data port
  //  ...and use Util method to convert to Fahrenheit
  val tempInF = Util.toFahrenheit(api.getcurrentTemp().get)
  // convert values of setpoint (saved local to the thread) to Fahrenheit
  val setPointLowInF = Util.toFahrenheit(setPoint.low)
  val setPointHighInF = Util.toFahrenheit(setPoint.high)

  // construct possible control message to send to Fan
  //  ...a Slang option type is used to indicate if a control message will be needed or no
  val cmdOpt: Option[FanCmd.Type] =
    // if input temperature is higher than high set point, then
    //  indicate a control message is needed (using `Some`) and the
    //  message is to turn the fan on to cool the environment
    if (tempInF.degree > setPointHighInF.degree) Some(FanCmd.On)
    // if input temperature is lower than low set point, then
    //  indicate a control message is needed (using `Some`) and the
    //  message is to turn the fan off to let the environment warm
    else if (tempInF.degree < setPointLowInF.degree) Some(FanCmd.Off)
    // ... otherwise no control message is needed
    else None[FanCmd.Type]()
  // use pattern matching on cmdOpt to selectively send message on `fanCmd` output event d
  cmdOpt match {
    // case where cmd has be built for sending
    case Some(cmd) =>
      // use the auto-generated port API to send the message
      api.sendfanCmd(cmd)
      api.logInfo(s"Sent fan command: $cmd")
    // case where no cmd was built
    case _ =>
      // don't send a command, just log
      api.logInfo(s"Temperature ok: ${tempInF.degree} F")
  }
}

// component local variable declared to hold received set point
var setPoint: SetPoint =
  SetPoint( ... <initialized set point value> ... )

// message handler for the `setPoint` input event data port -- `value` is the message payl
override def handlesetPoint(value : BuildingControl.SetPoint): Unit = {// this signature/s

  // ==== developer programs application code below ===
  api.logInfo(s"received setPoint $value")
```

```
  // assign received set point value to component local variable
  setPoint = value
}
```

While the message handler methods form the *developer's view* of the Compute Entry Point, behind the scenes, the HAMR-generated infrastructure for a sporadic thread includes a `compute` method that represents the *infrastructure's view* of the Compute Entry Point. The Slang version of this method is shown below.

**Note**

ToDo: The processing of *all* incoming events is not aligned with the AADL standard. Instead, only one event should be processed per dispatch. It also does not incorporate the more sophisticated semantics for "dispatch trigger" ports, urgency, etc.

```
def compute(): Unit = {
   // get ids of event / event data ports that have pending messages.
   val EventTriggered(portIds) = Art.dispatchStatus(TempControl_i_BridgeId)
   // "freeze" data ports -- move data port values from infrastructure to application sp
   Art.receiveInput(portIds, dataInPortIds)

   // === invoking application code (event handlers) ===
   // for each arrived event, call the corresponding event handler
   for(portId <- portIds) {
     // if an event arrived on the `fanAck` port
     if(portId == fanAck_Id){
       // get the message payload and call the fanAck event handler with the message pay
       val Some(BuildingControl.FanAck_Payload(value)) = Art.getValue(fanAck_Id)
       component.handlefanAck(value)
     } // process setPoint event data in same manner as above
     else if(portId == setPoint_Id){
       val Some(BuildingControl.SetPoint_Payload(value)) = Art.getValue(setPoint_Id)
       component.handlesetPoint(value)
     }
     else if(portId == tempChanged_Id) {
       // `tempChanged` port is event (not event data) so there is no payload to pass to
       component.handletempChanged()
     }
   }
   // after all event handlers have run, propagate to consumers the values that they wro
   Art.sendOutput(eventOutPortIds, dataOutPortIds)
 }
```

Similar to the way that the periodic thread application code ( `timeTriggered` method) was delimited by `receive input` and `send output` , the same form of delimiting occurs in the code above for the sporadic application code (the event handlers).

## Finalize Entry Point

The notion of Finalize Entry Point is somewhat under-specified in the AADL standard. In HAMR, the Finalize Entry Point is used to for any application logic needed to achieve a graceful shutdown of the application component. This might include close any system services (e.g., GUIs, communication links) not directly modeled in AADL, or sending shutdown logging messages. A HAMR thread will not receive any incoming port communication nor propagate any outgoing port communication during the execution of the Finalize Entry Point.

**Note**

ToDo: Need to indicate what system state / condition triggers the activitation of the Finalize Entry Point.

## AADL Computational Paradigm – Rationale for Thread Structure

The following clause from Section 5.4 (2) of the AADL summarizes some fundamental properties of AADL's computation paradigm:

### AADL Computational Paradigm (AADL Standard - Clause 5.4 (2))

AADL supports an input-compute-output model of communication and execution for threads and port-based communication. The inputs received from other components are frozen at a specified point, by default the dispatch of a thread. As a result the computation performed by a thread is not affected by the arrival of new input until an explicit request for input, by default the next dispatch. Similarly, the output is made available to other components at a specified point in time, for data ports by default at completion time or thread deadline. In other words, AADL is able to support both synchronous execution and communication behavior, e.g., in the form of deterministic sampling of a control system data stream, as well as asynchronous concurrent processing.

The phrasing "AADL supports" indicates that AADL emphasizes the model of computation, but it does not require it. It emphasizes it by:

- defining a constrained structure for threads and thread states that support the model,
- defining a collection of run-time services sending and receiving values over components ports,
- defining a collection of built in properties for configuring and specifying properties (including timing and latency properties) that rely on the model, and
- providing foundations for automated schedulability and latency analysis that rely on the model.

The computational paradigm described in the standard clause above has long been emphasized within the real-time systems community as an approach to engineer systems that are more easily analyzable. For example, the following pseudo-code suggesting a structure for real-time tasks is taken from "Analysable Real-Time Systems Programmed in Ada" (Section 2.3 p. 30) by Alan Burns and Andy Wellings (the notation/syntax/comments have been adapted slighly to make a stronger connection to AADL terminology):

```
loop
  wait for dispatch
  // ==== RECEIVE INPUT =====
  Read data from other components into input port variables In1, In2, , Inm
  // ==== DO APPLICATION WORK ====
  // compute output values from current component
  // state S (may include many local variables) and input values
  Out1, Out2, , OutN := G(S,In1, In2, , Inm)
  // compute new local state from current local state and input values
  S := F(S, In1, In2, , Inm)
  // ==== SEND OUTPUT ====
  Write data in output ports to other components
end loop // (repeats forever)
```

While AADL and HAMR provide much more flexibility in how the application code (the section marked "DO APPLICATION WORK") is structured, the overall effect of AADL's computational paradigm is very similar to the Burns/Wellings outline above. When an AADL thread component is declared to be computational paradigm compliant (the notation for this is not finalized), HAMR goes beyond what is required in the AADL standard to support the computational paradigm – in particular, it generates infrastructure code and application code skeletons in way that guarantees that the application code will conform to AADL's computation model. Specifically, referring to the pseudo-code above, the HAMR approach hides from the developer the code for looping, waiting for dispatch, and sending/receiving information between components. This enables the developer to focus on coding and verifying the "application work" code.

Some of the key aspects in the HAMR "strict AADL adherence" approach are:

- constrained structure for threads and thread states – HAMR follows the entry point structure defined in the standard, and it adheres to the standard guidance regarding Compute Entry Point structure for periodic and sporadic components. It goes beyond what is specified in the standard to actually *enforce* the structure through its code generation patterns for periodic and sporadic Compute Entry Points.
- communictation – HAMR further constrains the AADL use of run-time services for sending and receiving values over components ports by created dedicated APIs for each component for sending and receiving values over ports.
- arrangement of communication wrt application code – while AADL provides some leeway as to when the exchange of information between components occurs, HAMR takes the strictest interpretation of the AADL standard and always forces port receives and sends to occur at the beginning and ending of the Compute Entry Point (as was illustrated in the auto-generated code above by delimiting the Compute Entry Point application code with `receiveInput` and `sendOutput` calls).

These HAMR restructions have the following implications:

- Input values are read once – there is no notion of getting updated input values in the middle of the task processing. This avoids issues related to race conditions and otherwise having application code use locks on data, which due to developer inexperience and mistakes can lead to concurrency related bugs. All locking or synchronization via scheduling is isolated in the infrastructure code (which can be assured once and for all).
- Input values are read together at the same conceptual point in time. This avoids inputs values being uncorrelated in time.
- Task application code never blocks while waiting for data or some other condition in the middle of its execution. This makes reasoning about worse-case execution time (and resulting scheduling) easier.

HAMR's strict interpretation of AADL facilitates compositional verification of components (this will be illustrated in forthcoming chapters on HARM AADL/Slang contracts). The constraints on the Compute Entry Point enable each invocation of the entry point to be view semantically as a function from port inputs to outputs (with optional update of local state variables – this means that the local state variables are both inputs and outputs to the function). This facilities designing HAMR AADL/Slang contracts based on pre/post-conditions and invariants on thread component entry points.

As indicated in HAMR Overview (ch00-hamr-overview.html#ch00-hamr-overview), it is possible to use AADL and HAMR in portions of the system that do not adhere to the AADL computational paradigm. In such cases, some of AADL's specification and analysis concepts will not apply (or rather, they can be applied to achieve some utility, but they will not be sound). The most common use-case for such situations is the incorporation of legacy components.

**Note**

**ToDo: Give point to section of documentation discussing dealing with legacy components:**

# Port-based Communication

The primary form of AADL inter-component communication is port-based communication (described in the section below), which covers most of the communication patterns used in embedded and distributed systems. AADL also provides facility for ``subprogram calls" (which are roughly the same as remote procedure calls) and access to shared data objects. Among these, port-based communication is by far the most developed and emphasized in the AADL standard. *HAMR only supports port-based communication at this time*.

## Port Communication Run-Time Services and HAMR Application API Concepts

XXXXXXXX stopped here XXXXXXXXXXX

- Figure - of receive input, send output and quotes from standard
- Notion of port states
- Receive input and send output as abstractions of real-time middleware
- Component-specific HAMR generated APIs as wrappers for generic middleware concepts

## Port Categories: Data, Event, and Event Data

AADL provides three categories of port-based communication: event data ports, data ports, and event ports. The purpose of AADL's three categories of ports is best described in Clauses (3, 12, 13, 14) of Section 8.3 the standard (quoting from the standard below).

### AADL Port Categories (AADL Standard - Section 8.3)

3. AADL distinguishes between three port categories. Event data ports are ports through which data is sent and received. The arrival of data at the destination may trigger a dispatch or a mode switch. The data may be queued if the destination component is busy. Event data ports effectively represent message ports. Data ports are event data ports with a queue size of one in which the newest arrival is kept. By default arrival of data at data ports does not trigger a dispatch. Data ports effectively represent unqueued ports that communicate state information, such as signal streams that are sampled and processed in control loops. Event ports are event data ports with empty message content. Event ports effectively represent discrete events in the physical environment, such as a button push, in the computing platform, such as a clock interrupt, or a logical discrete event, such as an alarm.

12. Data ports are intended for transmission of state data such as sensor data streams. Therefore, no queuing is supported for data ports. [...]
13. Event data ports are intended for message transmission, i.e., the queuing of the event and associated data at the port of the receiving thread. A receiving thread can get access to one or more data element in the queue according to the Dequeue_Protocol and Dequeued_Items properties (see Section 8.3.3). [...]
14. Event ports are intended for event and alarm transmission, i.e., the queuing of events at the port of the receiving thread, possibly resulting in a dispatch or mode transition. A receiving thread can get access to one or more events in the queue according to the Dequeue_Protocol and the Dequeue_Items property. [...]

XXXXXXXX stopped here XXXXXXXXXXX

In listing XXX, the port `fanCmd` in `TempControl` that sent a command to turn the fan on/off was modeled as an `event data` port because it is naturally aperiodic (the command doesn't change with each temperature sensor reading) and it is natural to trigger a dispatch of the `Fan` component only when it needs to change its state (on/off). The `currentTemp` reading output by the `TempSensor` was modeled as a `data` port so that the latest temperature reading would always be available to the `TempControl` component, but the arrival of data to `TempControl` need not trigger a dispatch. A notification from the `TempSensor` indicating that the temperature changed was modeled as an `event` port (it communicates a message with no payload). A reasonable alternative design could model the `currentTemp` as an event data port, which would produce an aperiodic message only when the temperature changed (removing the need for the separate `tempChanged`). With `currentTemp` as an event data port, it is also possible to have the `TempSensor` message sent periodically at the same rate of the `TempSensor` thread, and this would cause the `TempControl` component to be dispatched at the same rate. alternatives.

HAMR supports all three types of ports above.

In HAMR's support for ARINC 653-like platforms, these are implemented using ARINC 653 sampling ports. For the seL4 micro-kernel, HAMR implements these using shared memory with capability configurations that only allow writing by the sender and reading by the receiver.

AADL provides property sets to configure different types of policies related to ports (e.g., specifying size of event data port queues, what happens when event data port queues are full, etc.).

> **Note**
>
> **ToDo:** discuss port properties.

## Port Directionality: In and Out

AADL supports three different directional modalities (emph{in}, emph{out}, and emph{in out}) for ports (quoting from clauses of Section 8.3 of the standard):

```
(7) Ports are directional. An out port represents output provided by the sender, and an ir
```

HAMR *does not* support `in out` ports. We view such ports as an unnecessary complication as their effect can be realized via a pair of an `in` port and an `out` port. In most analysis or code generation capabilities, {bf in out} ports would be split `behind the scenes'' anyway into distinct ``in` and `out` ports. Furthermore, information flow specification and analysis, interface contracts, etc. are all simpler when ports are explicitly unidirectional at the model level.

## Port Connection Fan In / Fan out

# Computational Model

subsubsection{Application programmer's view of ports}

The programmer's view of ports and the specific representation of ports in the programming model is an important issue. Relevant clauses from the standard are given below.

begin{quote} (10) Ports appear to the thread as input and output buffers, accessible in source text as port variables.

1. [...] From the perspective of the application source text, data ports are accessible in the source text as data variables. From the perspective of the application source text, event ports represent event queues whose size is accessible. Incoming events may trigger thread dispatches or mode transitions, or they may simply be queued for processing by the recipient. From the perspective of the application source text, event data ports represent message queues whose content can be retrieved.

16. Data, events, and event data arriving through incoming ports is made available to the receiving thread, processor, or device at a specified input time. For a data port the input that is available through a port variable is a data value, while for an event or event data port it can be one or more elements from the port queue according to a specified dequeuing protocol (see Section 8.3.3). From that point on any newly arriving data, event, or event data is not available to the receiving component until the next dispatch, i.e., the content of an incoming port that is accessible to the application code does not change while the thread completes its execution.

(17) By default, port input is frozen at dispatch time. For periodic threads or devices this means that input is sampled at fixed time intervals. end{quote}