Model-Driven Development for the seL4 Microkernel Using the HAMR Framework

Jason Belt, Robby, John Hatcliff[1]

Kansas State University

John Shackleton, Jim Carciofini, Todd Carpenter

Adventium Labs

Eric Mercer

Brigham Young University

Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind

Collins Aerospace

Ihor Kuz[2], Kent Mcleod[2]

Kry10 Limited

Abstract

Verified microkernels such as seL4 provide trustworthy foundations for safety- and security-critical systems. However, their full potential remains unrealized due, in part, to lack of application development environments that help engineers integrate the microkernel's configuration and hosting of application code with modeling, analysis, and verification tools that address broader aspects of the development lifecycle.

This paper presents a model-driven tool chain for the seL4 microkernel based on the open source High Assurance Modeling and Rapid engineering (HAMR) code generation framework for the Architecture and Analysis Definition Language (AADL). We describe how the semantics of AADL communication and threading can be realized in terms of the access primitives and strong spatial and temporal partitioning mechanisms provided by seL4. For AADL users, seL4 provides a high-assurance platform with formally verified enforcement of component boundaries and communication pathways. For seL4 users, AADL provides high-level abstractions for developing seL4 applications, along with an ecosystem of system engineering and analysis tools. We illustrate the framework by applying a model-based development environment for increasing resiliency against cyber attacks to an unmanned aircraft flight control system.

[1]Corresponding author.
[2]This work was performed while Kuz and Mcleod were employed at CSIRO's Data61.

Preprint submitted to Elsevier *January 22, 2022*

1. Introduction

Modern high-assurance separation kernels provide trustworthy foundations for safety- and security-critical systems. Unlike traditional large kernels and operating systems that perform arbitrary functions, "a separation kernel's primary security function is to partition (separate) the subjects and resources of a system into security policy-equivalence classes, and to enforce the rules for authorized information flows between and within partitions" [22].

Separation *microkernels* have minimal and highly optimized functionality that allows fine-grained configuration of system policies. This allows these microkernels to form a dependable base for building different types of systems. For example, separation microkernels can provide strong time and space separation between user-level applications, provided the system is configured appropriately. Such separation can prevent compromises or faults in one partition from compromising or interfering with other functionality hosted on the system. This supports non-interference, which can be relied on as a fundamental property by upper levels of the system architecture [25].

This highly focused functionality makes separation microkernels amenable to high-assurance design practices. Early microkernels relied heavily on process- and testing-based approaches for assurance. The implementation of the more recent high-assurance, high-performance seL4 separation microkernel is formally (mathematically) proven correct against its specification, and has been shown to enforce strong security properties, including spatial separation.

Given such a high-assurance foundation, developers can build mixed criticality systems that do not require all the components to be certified to the level of the highest criticality component. In addition, components may be updated throughout a system's lifecycle (e.g., to add or update functionality). Depending on the system design, a strong separation foundation can facilitate modification and recertification of the updated component, without necessarily requiring recertification of all the other components.

However, the nature of separation microkernels, in particular the highly focused feature set, means that they often provide only lower-level configuration mechanisms, more oriented to memory blocks, processes, functions, etc. All other functionality must be provided at the user-level. Therefore, even though the use of separation microkernels provides an extremely solid foundation for high assurance, the separation principles and guarantees provided by the kernel are not sufficient; engineers and evaluators face significant challenges in realizing correct application logic, security policies, and system engineering when building on top of the kernel.

For example, in 2007 the US National Information Assurance Partnership (NIAP) published the Separation Kernel Protection Profile (SKPP). The SKPP specifies the security functional and assurance requirements for a class of separation kernels. Shortly thereafter in 2011, the NIAP sunsetted the SKPP, stating, "...conformance to this protection profile, by itself, does not offer sufficient confidence that national security information is appropriately protected in the context of a larger system in which the conformant product is integrated. Designers of such systems must apply appropriate

systems security engineering principles and techniques to afford acceptable protection for national security information. In particular, it is the responsibility of the system designer and authorized administrator to define support for a coherent application-level security policy in the separation kernel's configuration data, as well as to ensure that the configuration data itself is coherent and self-consistent" [22]. In addition, application of the separation kernels had a high learning curve that was exacerbated by the limited tool support available at the time.

To help engineers use seL4, the seL4 team introduced the Component Architecture for microkernel-based Embedded Systems (CAmkES) – a domain-specific language for configuring seL4 using component-oriented idioms. In CAmkES, developers can define kernel enforced system partitions as components, controlled access to partitions via ports with read/write access policies, and information flows between partitions as connections between component ports. The CAmkES specification can then be translated to lower-level capability descriptions that are used to configure seL4.

Multiple United States (US) Department of Defense (DoD) research projects that emphasize high-assurance have built tool chains that utilize seL4 and CAmkES [7]. A principal goal of the Cyber Assured Systems Engineering (CASE) project sponsored by the US Defense Advanced Research Projects Agency (DARPA), in which the authors have participated, has been to develop model-driven development tools that use the industry standard Architecture Analysis and Design Language (AADL) [10, 16, 9] to help engineers make systems more resilient to cyber-attacks. This was accomplished by carefully modeling desired system architectures and information flows in AADL, analyzing the system for security vulnerabilities, and then generating system infrastructure code and kernel configurations (using CAmkES) for deployment on seL4. The beneficial properties of microkernels described above were key to overall assurance arguments and cyber-resiliency.

Implementing the tool chain to support CASE goals posed a number of engineering challenges. For example, CAmkES threading and communication primitives were not aligned with the semantic primitives in AADL that were utilized by other CASE tools for formal analysis and verification. The baseline CAmkES did not support true one-way communication, which was needed to achieve precise reasoning about information flow controls for satisfying CASE cyber-assurance objectives. In addition, there were significant gaps between the seL4 artifacts and other systems engineering activities including hazard analysis, security analysis, schedulability analysis, traceability to requirements, and formal specification of component-level and system-level functionality.

In this paper, we describe a model-driven development tool chain for seL4 that addresses the issues described above. The tool chain is based on High-Assurance Model-based Rapid engineering for embedded systems (HAMR). HAMR provides analysis, verification, and code generation capabilities for system architectures defined in the SAE standard AADL [27].

By extending HAMR to generate code that targets the seL4 CAmkES runtime framework and build frameworks, we enable seL4 developers to leverage AADL's high-level modeling abstractions as well as AADL's ecosystem of systems-engineering timing, safety, and security analysis and verification capabilities. HAMR automatically connects the low-level seL4 configuration to a model-driven development paradigm

that enables engineers to work with rich architecture descriptions. Since HAMR generates analyzable artifacts for each step in the process, developers can be confident that the architecture described in the models is correctly enforced by the generated seL4 configuration.

The specific contributions of this paper are as follows.

- We describe how the communication and threading semantics of AADL (which is designed to facilitate analysis and verification) is mapped onto constructs provided by the CAmkES seL4 configuration language.
- We present enhancements of CAmkES and associated HAMR-generated infrastructure code to support high assurance communication semantics, such as one-way communication between components. Such communication maintains separation and supports formal reasoning about information flow controls required in many high-assurance embedded systems.
- We describe how CAmkES threading and the seL4 domain scheduler mechanism are extended to support the periodic and sporadic real-time tasking of AADL, including the time-partitioning of tasks in static schedules.
- We illustrate how the HAMR code generation architecture is enhanced to generate C code from AADL models that is compatible with seL4 and CAmkES build environments. This includes code generation support for AADL Base Types and user-defined types specified using the AADL Data Model Annex.
- We discuss HAMR and seL4 infrastructure for deploying system builds on both the QEMU hardware emulator and development boards. This significantly eases prototyping and debugging of seL4-based applications.
- We describe our experience with the framework on the DARPA CASE program. This includes a detailed discussion of how HAMR and seL4 are used to generate deployable code for a mission control system for unmanned aircraft. This assessment is supported by a performance evaluation of the tool chain and resulting code.

The HAMR framework is being applied by multiple industry partners in projects funded by the US Army, Air Force Research Laboratory (AFRL), DARPA, and the Department of Homeland Security (DHS). The HAMR implementation and examples described in this paper are available under an open-source license [3].

2. Model-Driven Development Approach

2.1. Overview

HAMR is a multi-platform code generation framework that supports system implementations for Javascript, Java Virtual Machine (JVM) using Slang (a safety-critical

[3] Source code, video tutorials, and supporting documentation for the HAMR distribution are available at [33]. Code repositories for examples in the paper are referenced in Section 8

subset of Scala) [24], Linux, and seL4 platforms (see [12] for an overview of HAMR's multi-platform capabilities). The seL4 platform has been the primary platform used on the DARPA CASE program, although JVM and Linux were also used in a systematic prototyping workflow.

On CASE, HAMR is an element of a broader suite of tools called BriefCASE, developed by Collins Aerospace and its collaborators (see [6] for a detailed overview, which we summarize here). BriefCASE is predicated on a Model-Based Systems Engineering (MBSE) process, in which models are the primary vehicle for communication and understanding among the parties tasked with designing the system. Furthermore, BriefCASE Model-driven Developent (MDD) models are the primary design artifacts used for analysis, verification, testing, and code generation.

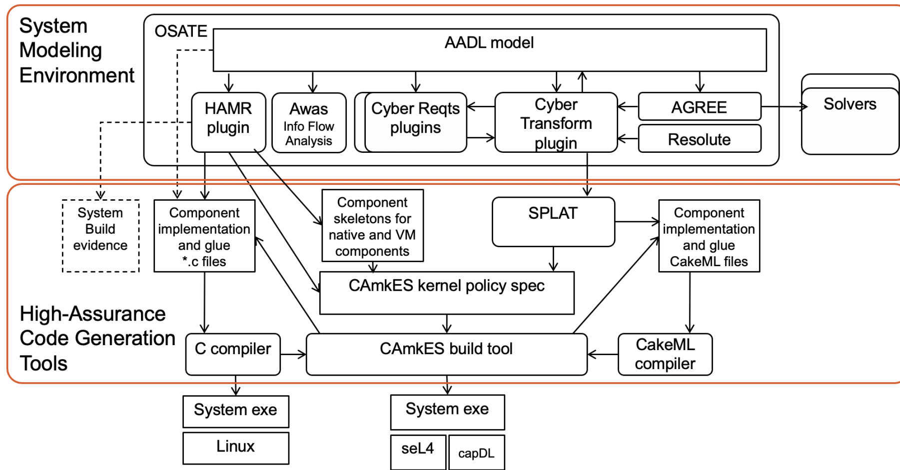

Figure 1: BriefCASE tool architecture and workflow.

The BriefCASE tool workflow (see Figure 1) starts with the development of a baseline AADL model of the system architecture focusing on the desired functionality. BriefCASE is implemented as a collection of plugins in the Eclipse-based Open Source AADL Tool Environment (OSATE), the flagship AADL modeling tool. In OSATE, the AADL system model can be analyzed using any of the supported AADL tools (e.g., resource usage, information flow, latency, and schedulability). The Awas AADL information flow analysis tool [29, 32] was especially important for CASE since it enabled model-level system information flows to be interactively browsed and queried to both discover problematic flows and (during iterative cyber-resiliency hardening of the design) confirm that appropriate information controls are in place. BriefCASE integrates cyber-vulnerability analysis tools [18, 23] (Cyber Reqts plugins in Figure 1) that analyze the architecture model for cybersecurity vulnerabilities and generate requirements that, when addressed, will mitigate those vulnerabilities. These requirements are imported into BriefCASE, and associated claims about the system are managed by the Resolute assurance case tool [3].

As requirements are addressed in the design, and subsequently the implementa-

tion, an assurance case is updated with corresponding evidence, often captured directly from the model (e.g., establishing that partitioning or information flow control properties are satisfied) or by supporting analysis and verification tools (e.g., establishing that functional properties expressed as formal behavioral specifications on components are satisfied). For certain classes of requirements associated with vulnerabilities uncovered by BriefCASE analysis, BriefCASE provides automated architecture transformations (Cyber Transform plugin) to "harden" the system design against the vulnerabilities. Some of these transformations involve automatically inserting *cyber-resiliency components* such as data filters or behavior monitors. In many cases, the behavior of the components is formally specified in the AGREE [8] AADL contract language, and AGREE compositional model-checking (using underlying SMT solvers) is used to establish that the system meets certain end-to-end system-level properties. Given a sufficiently detailed formal specification of cyber-resilient component behavior, the Semantic Properties of Language and Automata Theory (SPLAT) tool [11] generates code, as well as proofs showing that the generated code is correctly compiled and meets its specification [20]. SPLAT generates code in the CakeML [17] dialect of Standard ML [21], and CakeML component implementations are compiled to executables using the CakeML verified compiler [17].

HAMR OSATE menu options and its code generation pipeline are wrapped in an OSATE plugin. To support development of new components, HAMR generates code skeletons for AADL thread components and APIs for AADL port-based communication. These artifacts are platform-independent and enable application code for threads to be easily deployed on Linux or seL4. HAMR generates seL4 kernel configuration information, including artifacts for the seL4 CAmkES build system. HAMR generates AADL run-time infrastructure to realize AADL semantics for threading and communication in terms of lower-level CAmkES and seL4 artifacts. For AADL process components that are specified as Virtual Machine (VM) based components, HAMR generates code for integrating non-VM and VM components, and generates a VM configuration including device table mappings to realize communication over cross-VM-boundary pathways modeled as AADL ports. In addition, HAMR performs key steps in the system build process by generating CMAKE configurations for generated C code, generating wrapper code to enable the integration of CakeML-based components, and creating the directory structure and scripts expected by CAmkES. Once all of these artifacts (including the implementation of all components) are in place, the CAmkES build framework is run to generate a deployable build for seL4. The build can be deployed in the QEMU hardware emulation environment or directly on a development/product board.

While this paper focuses on C code generation for the seL4 platform, CASE developers have also made use of HAMR's other backends for prototyping and simulation. In a typical use case, engineers construct initial AADL models, and component functionality is mocked up using Slang [24]. This enables design and testing of system messages types, planning of integration, etc. Moreover, the system prototype can be executed on the JVM or compiled to JavaScript (using the ScalaJS tool). While these deployments do not provide precise real-time behavior, HAMR provides a number of interesting simulation and visualization capabilities for its JVM implementations that are useful for rapidly gaining an understanding of system execution. Once such proto-

types are in place, the engineers often regenerate the system build for a C-based Linux deployment. Since Slang can be compiled to efficient C code, in some cases Slang is translated automatically to C, and in other cases, component implementations are written from scratch in C (e.g., when interfacing with legacy C code or when bringing in C libraries to be used in the full system). During these steps, the behavior of VM-based components and components that interface with system hardware are mocked up. Subsequently, the system can be regenerated for seL4 deployment running on QEMU, with legacy code wrapped in Linux virtual machines in seL4 partitions (the seL4 infrastructure was extended so that QEMU can support emulation of seL4-hosted virtual machines), and different forms of system testing can be conducted. Finally, the system can be deployed on bench and product hardware, once remaining code used for interfacing with external hardware is added. System integration testing can then be performed.

The BriefCASE environment, including the HAMR tool chain and QEMU emulator, is publicly available and can be installed natively on Mac OS, Linux, or Windows platforms, or in a virtual machine running Linux using Vagrant virtual machine setup scripts [33]. Currently, SPLAT can only be run on Linux; however we expect this limitation to be removed in future versions of the tool. The Vagrant-based virtualization capability is also used to support regression testing and continuous integration for the HAMR pipeline, including the ability to script the build process, as well as automatically deploy and test seL4-based system executions using QEMU emulation.

2.2. Key Concepts

The component-oriented emphasis of AADL modeling and the use of seL4 to provide partitioning of components is key to the Collins Aerospace CASE approach. Together these technologies provide the ability to (a) design and deploy a system as a collection of distinct strongly isolated units with controlled information flow between them, and (b) easily rearrange the units and associated information flows to incorporate cyber-resiliency components that fulfill overall system security requirements. In the remainder of this section, we provide an overview of two of the BriefCASE technologies that highlight the approach.

The Awas [29, 32] AADL information flow analyzer and visualizer enables developers and auditors to understand, reason about, explore, and visualize system dependencies and information flows at scale across components and sub-systems. Awas processes the AADL system architecture model, specifically its inter-component connection descriptions and intra-component flow specifications, to provide formal system-wide impact and flow analyses. Such flows include component data/control flows, security-oriented information flows, and fault/error propagation specified using by the AADL Error Modeling Annex (EMv2). Awas also provides a user-friendly Domain Specific Language (DSL) to query, check, and visualize custom safety/security system properties.

Figure 2 illustrates example output from Awas information flow reachability analysis. Working with our UAV example (presented in Section 4), the output illustrates how Awas analysis might help an analyst with the following design concern: “How does map information propagate from Ground Station to UAV and through UAVs Mission

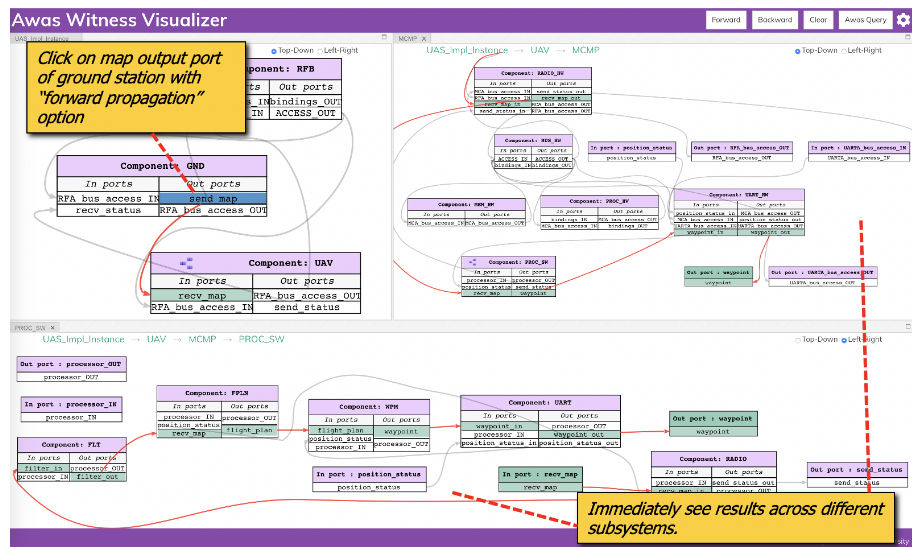

Figure 2: Awas AADL Information Flow Analysis.

Computer to produce a waypoint?" Awas can compute and visualize how the information in Ground Station messages flows through the system as well as the components or ports that may directly or indirectly consume data derived from that information. Awas supports a number of forms of forward and backward interactive information queries. Using the Awas script-based query language, one can specify and check more complex properties, e.g., that information must flow through specific ports or components. These end-to-end flow specifications are often useful for supporting verification of the effectiveness of cyber-resiliency components. An Awas specification can state that information from untrusted components such the Ground Station always flow through the attestation gate and filter components before reaching the Flight Control or other components that make critical decisions about the flight or mission of the UAV.

One of HAMR's primary contributions to the DARPA CASE program goals is that its code generation for seL4 can ensure that the component partitioning and inter-component information flows (as specified using one-way AADL connections) is precisely reflected in the deployed system: all such information flows present in the model and Awas analysis are present in the deployed system, and the deployed system does not contain any additional information flows between modeled components beyond those present in the model.[4] This strong guarantee, combined with the formal proof of

[4]HAMR and seL4 only provide assurance for AADL connections (inter-component information flows). Intra-component flows, modeled as AADL flow specifications, correspond to information flows through source code statements. AADL flow specifications are processed and visualized by Awas, but they are not enforced at the source code level in BriefCASE. Code-level slicing and information flow tools can potentially be used to support this. Our previous work for SPARK Ada information flow [2, 1, 28] would be an especially appropriate foundation for information flow reasoning in Slang-implemented components.

correctness of seL4, means that for each communication channel between components (corresponding to the component partitioning and information flows specified in the AADL model), no other component or actor in the system can access the channel or modify the contents of the channel.

Another key BriefCASE technology facilitated by HAMR's AADL code generation for seL4 is the semi-automated insertion of cyber-resiliency components into the architecture. Currently, the following transformations and component insertions are supported (most of these are illustrated in Section 4):

- *Filter* – Blocks messages that do not conform to the given specification
- *Monitor* – Detects violations of a given run-time condition and generates an alert
- *Switch* – Used with a Monitor to block messages when an alert is generated (also referred to as a *gate*)
- *Attestation* – Performs a measurement on non-local software to assess its trustworthiness
- *Virtualization* – Isolates software component(s) in a virtual machine
- *Proxy* – Inserts a pair of components to allow inspection of HTTPS message payloads
- *seL4* – Transforms the model to comply with seL4 component properties by isolating a thread in a process, with a binding to a processor (indicating isolation within an seL4 partition).

Because AADL's component-based modeling idioms cleanly architect the system into distinct units with explicit information flows *and* because seL4 guarantees the separation of components and the tamper-proof realization of one-way communication between components (no unanticipated backflows), these types of transformations (in both the model level and deployed system) become easy to achieve, whereas they would be very difficult if not impossible to achieve with the same levels of assurance in conventional system development approaches.

3. AADL

AADL [10, 16, 9] provides the modeling framework for the MDD tool chain described above. We present the portions of AADL treated by HAMR using a simple example (based on the description in [12]) and then present an "at scale" example from the CASE program in Section 4.

3.1. AADL Modeling Concepts

SAE International standard AS5506C [27] defines the AADL core language for expressing the structure of embedded, real-time systems via definitions of components, their interfaces, and their communication. In contrast to the general-purpose modeling diagrams in UML, AADL provides a modeling vocabulary of common embedded

software elements, hardware, and execution resources. The categories of software components are data, subprogram, subprogram group, thread, thread group, and process. Execution platform component categories that represent computing hardware are processor, virtual processor, memory, bus, virtual bus, and device (which is used to model sensors, actuators, or custom hardware). An AADL `system` component represents an assembly of interacting application software and execution platform components. Each category of component has a different interpretation when processed by AADL model analyses, and each category has a distinct set of standardized properties associated with it that can be used to configure the component's semantics or implementation.

For HAMR code generation, thread components are the most important since they determine the structure of code templates and APIs generated for communication. Typically, thread group, process and system components (representing both subsystems and the "top level" system) are used to aggregate threads and specify partitioning. In the AADL standard, thread components represent a schedulable unit of concurrent execution, while process components represent address spaces from which threads execute. In general, an AADL process may contain multiple threads, while a thread instance may only reside within a single process. When HAMR is used in workflows with seL4 as the intended target, process components are used to specify kernel *spatial partitioning* – each process represents a protected memory region that can only be accessed through APIs corresponding to the declared AADL ports for the component. Currently, when generating code for seL4, HAMR only supports processes containing a single thread. This restriction derives from the current threading approach used in the CAmkES seL4 kernel configuration framework, which HAMR uses to generate seL4 code. The BriefCASE vulnerability analysis tools process some of the AADL execution platform components such as processor, bus, etc. when performing various forms of security analysis for the broader system architecture. However, HAMR currently ignores these component categories when generating code.[5]

A *feature* is a part of an AADL component type definition that specifies how that component interfaces with other components in the system. In typical embedded systems, AADL *ports* are the most commonly used class of features. A port can be classified as either an *event port* (typically used to model interrupt signals or other notification-oriented messages without payloads), a *data port* (usually employed to model shared memory between components or distributed memory services where an update to a distributed memory cell is automatically propagated to other components that declare access to the cell), or an *event data port* (typically used to model asynchronous messages with payloads as are commonly found in publish-subscribe frameworks). Inputs to event ports and event data ports are buffered, and the size of the buffers as well as overflow policies can be configured using a standard set of AADL properties. Inputs to data ports are not buffered; newly arriving data overwrites the previous value. We use the terms *event-like* ports to refer to both event and event data ports

[5]For AADL software components, HAMR also ignores subprogram and subprogram group components because we chose to capture code structuring at the level of methods, classes, etc. in the code itself rather than at the model level. Furthermore, data components are only used to define data types for messages exchanged in port-based communication between components, following the approach outlined in the AADL Data Modeling Annex [26].

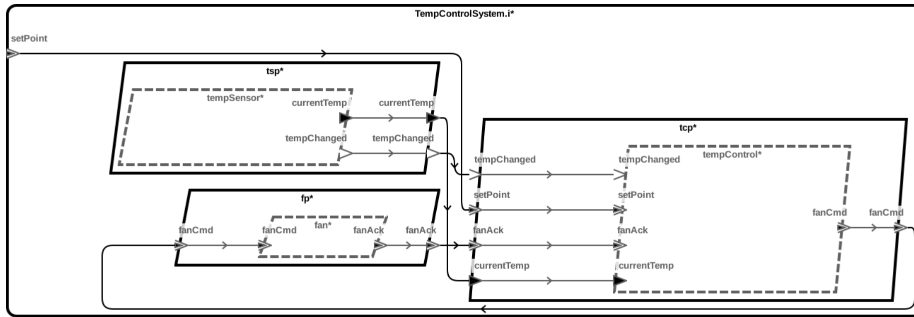

Figure 3: Temperature Control Example (excerpts) – AADL Graphical View

(which are both buffered) and *data-like* ports to refer to both data and event data ports (what both transmit data values). Although AADL provides bi-directional ports, due to its emphasis on information flow analysis and control, HAMR only supports unidirectional ports. This enables a clear graphical visualization and simpler information flow analysis.

HAMR autogenerates infrastructure code that implements the semantics of AADL port communication. It generates platform-independent APIs that hide the details of the underlying communication code. The APIs are then used by the thread application code for reading and writing to ports. Section 7.3 explains how CAmkES, along with additional auto-generated infrastructure, is used to implement AADL port communication. The code generation strategy uses read/write permission specifications in CAmKES to configure the seL4 kernel to enforce the directionality of AADL ports.

Figure 3 presents a portion of the AADL graphical view for a simple temperature controller that maintains a temperature according to a "set point" structure containing high and low bounds for the target temperature. As appropriate for the seL4 platform, each thread (dashed parallelogram) is isolated within a process component (solid parallelogram). The periodic `tempSensor` thread measures the current temperature and transmits the reading on its `currentTemp` *data port* (represented by a solid triangle icon). It also sends a notification on its `tempChanged` *event port* (represented by an arrow head) if it detects the temperature has changed since the last reading. When the sporadic (event-driven) `tempControl` thread receives a `tempChanged` event, it will fetch the value on its `currentTemp` *data port* and compare it with the most recent set points. If the current temperature exceeds the high set point, it will send `FanCmd.On` to the `fan` thread via its `fanCmd` *event data port* (represented by a filled triangle within an arrow head) to cool the temperature. Similar logic will result in `FanCmd.Off` being sent if the current temperature is below the low set point. In either case, `fan` acknowledges whether it was able to fulfill the command by sending `FanAck.Ok` or `FanAck.Error` on its `fanAck` event data port.

AADL provides a textual view to accompany the graphical view, and AADL editors such as OSATE ensure synchronization between the two. The listing below illustrates the *component type* declaration for the `TempControl` thread for the example above. The textual view illustrates that data and event data ports can have types for the data transmitted on the ports. In addition, properties such as `Dispatch_Protocol` and `Period` are used to configure the tasking semantics of the thread.

```
thread TempControl
features
 currentTemp: in data port Temperature.i;
 tempChanged: in event port;
 fanAck: in event data port FanAck;
 setPoint: in event data port SetPoint.i;
 fanCmd: out event data port FanCmd;
properties
 Dispatch_Protocol => Sporadic;
 Compute_Execution_Time => 10ms .. 10ms;
 Stack_Size=> TemperatureControl_Properties::StackSize;
end TempControl;

thread implementation TempControl.i
end TempControl.i;
```

Listing 1: AADL Textual View for TempControl Thread

The bottom of the listing declares an implementation named `TempControl.i` of the `TempControl` component type. Typically, AADL declarations of HAMR thread component implementations have no information in their bodies, which corresponds to the fact that there is no further architecture model information for the component (the thread is a leaf node in the architecture model). Using information in the associated thread type, HAMR code generation will generate platform-independent infrastructure, thread code skeletons, and port APIs specific to the thread, and a developer then implements the thread's application logic in the target programming language. The generated thread-specific APIs serve two purposes: (1) the APIs limit the kinds of communications that the thread can make, thus help ensuring compliance with respect to the architecture, and (2) the APIs hide the implementation details of how the communications are realized by the underlying platform.

The AADL core language can be extended with properties and annex sublanguages. Properties can be understood as named attributes (i.e., key/value pairs) that can be attached to a model element. AADL provides many pre-declared properties, and allows definition of new properties through user-defined property sets. In the listing above, `Dispatch_Protocol` is an example of a pre-declared AADL property that selects the dispatching semantics for a thread. HAMR supports AADL's `Periodic` protocol (which causes a thread to be dispatched at a regular interval) and the `Sporadic` protocol (which causes a thread to be dispatched on the arrival of messages on its event or event data ports). `Compute_Execution_Time` is a pre-declared property that allows the worse case execution time of a thread activation to be specified in the model. Scheduling tools such as the Adventium Labs FASTAR tool [30] can use this information to automatically support schedule generation and latency analysis. Other examples of pre-declared properties include communication properties (e.g., capturing queuing policies on particular ports, communication latencies between components, rates on periodic communication, etc.) and memory related properties (e.g., capturing sizes of queues and shared memory, latencies on memory access, etc.). For example, the `Stack_Size` is used to configure memory allocation in seL4. User-specified property sets enable one to define labels for various implementation choices available on underlying platforms (e.g., choice of middleware realization of communication channels, configuration of middleware policies, etc.).

The listing below shows the control subsystem specification for the `TempControl`

example. The operator interface, which sends set points to the temperature controller, is a separate component. Thus, the only feature declared on the component type (interface) is a port for the set point message. The component implementation declares named instances (e.g., `tsp`, `tcp`) of subcomponent processes. A processor instance `t_processor` is also declared, and the properties section includes a "binding" that associates the processes with the processor. The process subcomponents are "integrated" by declaring named connections (e.g., `ct`, `tc`) between the subcomponent ports. In contrast to the empty component implementations of the thread components, the system component implementation illustrates how architectural hierarchy is reflected in situations where a component implementation is not a "leaf" in the architecture model, but rather is realized as an integration of subcomponents.

`HAMR::Platform` is a user-defined property for HAMR code generations indicating which HAMR target platforms the model is suited for. In this case, in addition to seL4, the model can be used to target a number of other HAMR backends. `annex resolute` is an example of an AADL annex annotation indicating that the system implementation should be checked with the Resolute tool according to the `CASE_Tools` rules. A successful check implies that the AADL system instance generated from the system implementation specification is in the subset of AADL that can be processed by the CASE tool chain.

```
1  system TempControlSystem
2  features
3   setPoint: in event data port SetPoint.i;
4  end TempControlSystem;
5
6  system implementation TempControlSystem.i
7  subcomponents
8   t_processor: processor TempControlProcessor.i;
9   tsp: process TempSensorProcess.i {CASE_Scheduling::Domain => 2;};
10  tcp: process TempControlProcess.i {CASE_Scheduling::Domain => 3;};
11  fp: process FanProcess.i {CASE_Scheduling::Domain => 4;};
12 connections
13  ct: port tsp.currentTemp -> tcp.currentTemp;
14  tc: port tsp.tempChanged -> tcp.tempChanged;
15  fc: port tcp.fanCmd -> fp.fanCmd;
16  fa: port fp.fanAck -> tcp.fanAck;
17  sp: port setPoint -> tcp.setPoint;
18 properties
19  Actual_Processor_Binding => (reference (t_processor)) applies to tsp, tcp, fp;
20  HAMR::Platform => (JVM, Linux, macOS, Cygwin, seL4);
21     annex resolute {**
22        check CASE_Tools
23     **};
24 end TempControlSystem.i;
```

Listing 2: TempControl System Illustrating Subcomponents and Connections

AADL editors check for type compatibility between connected ports. HAMR supports data types declared using the AADL-standard Data Model Annex [26]. For example, the data type declarations associated with the temperature data structure are illustrated below.

```
1  data Temperature
2  properties
3   Data_Model::Data_Representation => Struct;
4  end Temperature;
5
```

```
6  data implementation Temperature.i
7  subcomponents
8   degrees: data Base_Types::Float_32;
9   unit: data TempUnit;
10 end Temperature.i
11
12 data TempUnit
13 properties
14  Data_Model::Data_Representation => Enum;
15  Data_Model::Enumerators=>("Fahrenheit","Celsius","Kelvin");
16 end TempUnit;
```

Listing 3: Data Types Declared Using AADL Data Model Framework

A standard property indicates that the `Temperature` type is defined as a `Struct` and the `Struct` fields and associated types are listed in the data implementation. The `degrees` field has a type drawn from AADL's standardized Base Types library. The `unit` field has an application-defined enumerated type.

HAMR also supports "raw" byte arrays as the data type for ports. This enables application component code to provide their own encoders and decoders of data formats, which can be useful when integrating legacy code.

3.2. AADL Application Code Execution Concepts

To help ensure that (a) system executables conform to AADL model semantics and (b) semantics are consistent across different AADL-aligned code generation frameworks, AADL defines principles for structuring application code and specifies key semantic steps in the form of Run-Time Services (RTS). AADL RTS are library functions, some of which are called by AADL infrastructure code while others are may be called by application code (e.g., to access values on component ports). The AADL RTS also provide an abstraction layer: details of the underlying platform execution may be hidden behind RTS APIs, allowing a significant portion of AADL application threading and infrastructure code to be platform independent. HAMR makes heavy use of these concepts to achieve its support for multiple platforms. This section summarizes these concepts, based on the presentation in [13] (see also [14]). Some of the most important tasks in the design of the HAMR seL4 code generator are to ensure that these semantics concepts are realized appropriately using CAmkES and seL4 capabilities.

According to the AADL standard, system execution is divided into phases: an *initialization* and a *compute* phase. During the initialization phase, platform services are launched, thread and communication infrastructure is set up, and each thread's application code may initialize thread local variables and put initial values on output ports. After all initialization activities are completed, the system moves to the "normal" compute phase in which thread application code is executed according to the configured scheduling policy. To support these distinct phases, the standard indicates that a thread's application code is organized into phase-related entry points (e.g., methods that are invoked from the AADL run-time) as illustrated in Figure 4. The *Initialize* entry point is called by the AADL run-time during the initialization phase, and the *Compute* entry point is called during the system's compute phase. The standard defines other entry points for handling faults, performing mode changes, etc., but HAMR only supports the Initialize and Compute entry points for seL4.

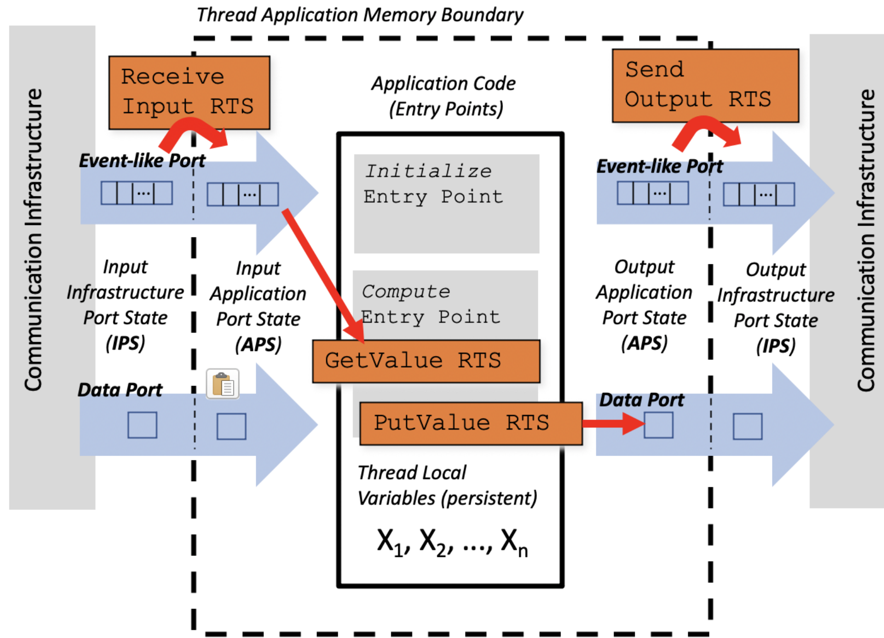

Figure 4: AADL Application Code Execution Concepts (adapted from [13])

In AADL terminology, "dispatching" a thread refers to its activation for execution. The thread `DispatchProtocol` property selects among several strategies for determining when a thread should be dispatched. In this paper, we consider only `Periodic`, which dispatches a thread when a certain time interval is passed, and `Sporadic`, which dispatches a thread upon arrival of messages to input ports specified as *dispatch triggers*.

Many of AADL's thread dispatching and execution concepts are based on long-established task patterns and principles for achieving analyzeable real-time systems [5]. Following these principles, with each activation of a thread, the application code of the thread will run to completion,[6] and for each activation, a thread abstractly computes a function from its input port values and local variables to output port values while possibly providing updated values for its local variables. To support this functional view, AADL specifies that input port values are "frozen" during execution. The presentation in [13] introduces the terms Infrastructure Port State (IPS) and Application Port State (APS) to distinguish between the communication infrastructure's and application code's view of ports. When a thread is dispatched, the component infrastructure uses the `ReceiveInput` Run Time Specification (RTS) to move one or values from queues and data port storage in the communication infrastructure IPS into the APS of the thread. This dequeues the values from the event-like IPS queues but leaves val-

[6]The AADL standard does allow preemption in entry points. However, for HAMR-based seL4 systems considered in DARPA CASE, scheduling and application code are designed so that thread Compute entry points *always* run to completion.

ues of the data ports IPS unchanged. Then the component application code is called, and the application's view of the ports (in terms of the APS) remains "frozen" as the code executes. This provides the application a consistent view of inputs even though queuing in the IPS may be concurrently updated behind the scenes. Throughout the Compute entry point execution, application code may read from the APS for a particular port using the `GetValue` RTS and write to the output APS for a particular port using the `PutValue` RTS. When the application code completes, the component infrastructure will call the `SendOutput` RTS to move output values from the output APS to the IPS, thus releasing the output values all at once to the communication infrastructure for propagation to consumers. This overall execution pattern means that, at the component's external interface, it follows the *Read Inputs; Compute; Write Outputs* structure championed by various real-time system methods (e.g., [5]) enabling analyzeability.[7] This pattern also enables the AADL Assume Guarantee Reasoning Environment (AGREE) contracts illustrated in Section 4 which specify component behavior as a relationship between inputs and outputs. For the Initialize entry point, no input ports may be read.

The representation of port states in terms of IPS and APS captures important aspects of the *semantics* of AADL port-based communication. However, the actual implementation of ports and communication may be optimized on particular platforms and under particular scheduling disciplines as long as the abstract semantics is maintained. For example, the presentation that we use for seL4 uses a single collection of shared memory locations to represented a producing thread's output IPS and connected consumer input IPS.

4. Example and Systems Engineering Concepts

This section summarizes an autonomous UAV-based surveillence system, developed on the DARPA CASE program, that we use to illustrate the full range of HAMR seL4 code generation capabilities. We first present an AADL model of the initial system, then discuss how we harden the system for cyber-resiliency using BriefCASE analysis and architecture transformations, and conclude with a walk-through of the hardened system architecture. The full system concept includes software and hardware for both a ground station and a UAV. For this discussion, we focus on the software subsystem on the UAV.

4.1. Initial UAV Software System Model

The AADL model of the initial UAV software is shown in Figure 5. The ports on the left-hand side of the top-level `SW.Impl` system component reflect the fact that (a) the UAV receives commands from a ground station to conduct surveillance over a geographical region (`radio_recv`); and (b) the UAV sends status messages

[7] The behavior described above is the canonical behavior emphasized in the AADL standard. The standard does allow for `ReceiveInput` and `SendOutput` to be called by the application code (not just the infrastructure) throughout the entry point execution. But this moves outside of the canonical semantics and makes analysis more difficult.

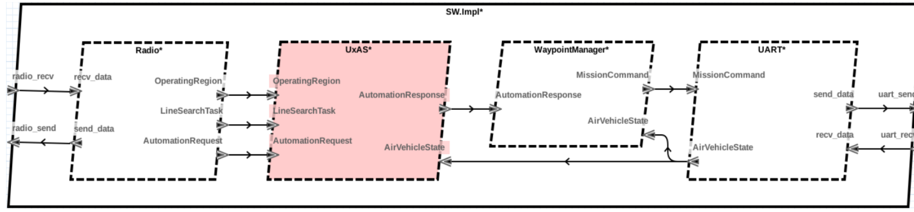

Figure 5: Software architecture for UAV surveillance system.

back to the ground station (`radio_send`). Based on the commands from the ground station, the UAV's on-board mission computer generates a flight plan. The ports on the right-hand side of the `SW.Impl` interface reflect the fact that waypoints from the generated flight plan are sent to the actual UAV flight controller through a Universal Asynchronous Receiver-Transmitter (UART) (`uart_send`). As the aircraft position changes, the flight controller will send updated position information back to `SW.Impl` (`uart_recv`).

There are four key components to the `SW.Impl` system (some of these contain their own hierarchy of subcomponents, but we only present the top-level interfaces in this figure): the radio driver (`Radio`), the autonomous flight planner (`UxAS`), the waypoint manager (`WaypointManager`), and the UART driver (`UART`).

The `Radio` component demultiplexes three types of messages from the ground station: `OperatingRegion`, `LineSearchTask`, and `AutomationRequest`. The `OperatingRegion` message defines a geographical area for UAV operation, including both *keep-in* and *keep-out* zones. The `LineSearchTask` message contains a set of waypoints corresponding to a geographical feature of interest (such as a river) over which the UAV must conduct its surveillance mission. The `AutomationRequest` message triggers the autonomous flight planner to generate a flight plan that satisfies the operating region constraints and enables the UAV to conduct surveillance on the feature of interest.

The `UxAS` component uses the Air Force Research Laboratory's OpenUxAS software to autonomously create flight plans for the UAV given an indicated line search task, operating region, and current location of the UAV (`AirVehicleState`) [15]. The resulting plan, which includes a collection of waypoints for the mission, is communicated downstream in an `AutomationResponse` message.

The `WaypointManager` component meters the full set of waypoints provided in the `AutomationResponse` to the resource-constrained flight controller. A small window of waypoints corresponding to the UAV's current location are sent to the controller within a `MissionCommand` message via the `UART` serial driver. As updates concerning the UAV's position are received from the flight controller and `UART` in an `AirVehicleState` message, the `WaypointManager` publishes the next set of waypoints in a new `MissionCommand` message.

4.2. Analysis, Requirements Generation, and Transformation

The initial model above is analyzed with the `GearCASE` and `DCRYPPS` BriefCASE plugins to identify potential cyber-vulnerabilities. Because `UxAS` is open-source,

third-party software, these tools identify it as potentially containing a supply chain vulnerability. Similarly, they identify a potential for the ground station being spoofed or compromised to send malicious messages to the UAV. These threat analyses are reviewed by the design engineers who then create new cyber-related requirements for the system. For example, for the supply chain vulnerability from `UxAS`, requirements are added to ensure that unverified or malicious code, which could potentially be embedded in the component, will not impact downstream components. Other cyber requirements are automatically generated and imported into the BriefCASE Requirements Manager after the threat analyses. These include four *wellformedness* requirements on message formats, two requirements for *monitoring* the behavior of `UxAS`, and an *attestation* requirement for ensuring the ground station has not been compromised. These new requirements are also automatically added to a `Resolute` assurance case. As the system is hardened and various verification tools are applied, Resolute incorporates evidence that assurance claims are met into the assurance case. `Resolute` analysis can be run at any time during development to determine which requirements are, and are not, currently supported by evidence.

Wellformedness requirements prevent malformed messages from causing buffer overrun or code injection attacks. These requirements are satisfied by augmenting the system architecture to include high-assurance *filter* components along key information pathways. Filters do not pass messages that are not wellformed. BriefCASE provides automatic transforms to insert filters at the point of a selected connection. Once a filter component is inserted, engineers define the meaning of `wellformed` for the filter with an AGREE contract.

Monitor requirements prevent unintended component behavior from affecting overall system functionality. These are addressed by augmenting the system architecture with software monitor components that observe key connections in the model, and raise alarms and filter messages to downstream components if suspicious behavior is detected. BriefCASE provides an automatic transform to add a monitor, and once added, engineers define the monitor behavior with an AGREE contract.

Attestation requirements prevent identity spoofing on external connections. These are satisfied by augmenting the system architecture with attestation components to validate identity and to apply filtering to only pass messages from known, and trusted, sources. As before, BriefCASE performs the transforms and engineers provide the AGREE contract defining the behavior of everything but the actual attestation software, which is a pre-packaged trusted component included with BriefCASE.

4.3. Model for the Hardened Software System

The transformed cyber-hardened model for the UAV surveillance system is shown in Figure 6. The high-assurance components inserted by BriefCASE are shown in green. The `AttestationManager` and `AttestationGate` ensure the UAV cannot receive malicious commands from a ground station running compromised software. The three filters before `UxAS` prevent it from receiving malformed messages. Since the `UxAS` software may not be trustworthy, its output is both filtered and monitored. The `GeoFence_Monitor` detects when mission waypoints are outside any of the keep-in zones or inside any of the keep-out zones, and the `Response_Monitor` detects when `UxAS` acts outside the command of the ground station.

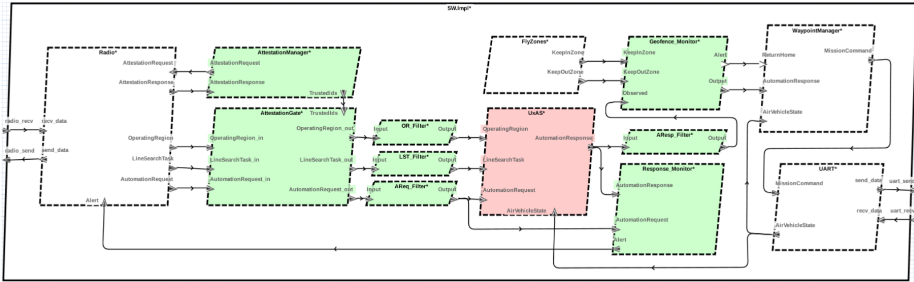

Figure 6: Cyber-resilient software architecture for UAV surveillance system.

A key aspect of the BriefCASE workflow is the formal specification of the high-assurance component behaviors, such as the filter and monitor policies. As an example, the AGREE contract specifying the behavior of the `GeoFence_Monitor` is shown in Listing 4. Here the `Policy` holds if all the mission waypoints observe the keep-in zones, observe the keep-out zones, and do not include duplicates. The `alert` is raised if the policy is ever violated and is used to trigger the `WaypointManager` to execute a pre-loaded "return to base" flight plan. The `output` is only sent if the `Policy` holds and the `alert` is low. The AGREE model-checking tool is used to prove that system-level cyber-requirements hold, and the verification utilizes the component level properties above. For the filter, monitor, and gate components, BriefCASE uses the `SPLAT` tool to automatically synthesize provably correct implementations that satisfy the given AGREE contracts. HAMR is then used to build the system, emitting proof artifacts that the build is a faithful representation of the model. The memory protection mechanisms of seL4 guarantee that the high-assurance code cannot be tampered with or bypassed in the actual deployment. Assurance evidence from each step of these evaluation, synthesis, and build activities are pulled into `Resolute` to support the overall cybersecurity assurance case.

5. HAMR Code Generation Architecture

In [12], we gave a brief overview of HAMR's code generation architecture. This section instantiates that description for the HAMR seL4 backend. Figure 7 illustrates the main concepts of HAMR code generation for seL4. Working off of an AADL *instance model* as generated by OSATE, HAMR generates a CAmkES specification of the deployment topology and other kernel configuration information (Section 7.1, Listings 14 and 15). For each AADL thread, HAMR generates infrastructure code that implements the AADL thread dispatch semantics. This includes: (a) *infrastructure code* for linking entry point application code to seL4 underlying scheduling framework (Section 7.2), for implementing the storage associated with ports, and for realizing the buffering and notification semantics associated with event and event data ports (Section 7.3); and (b) *developer-facing code* including thread code skeletons in which the developer will write application code, and *port APIs* that the application code uses to send and receive messages over ports (Section 6). For communication associated with

```
const is_latched : bool =
 Get_Property(this, CASE_Properties::Monitor_Latched);

eq Policy : bool = event(observed) =>
     (WAYPOINTS_IN_ZONE(GET_MISSION_COMMAND(observed), keep_in_zones)
 and WAYPOINTS_NOT_IN_ZONE(GET_MISSION_COMMAND(observed), keep_out_zones)
 and not (DUPLICATES_IN_MISSION(GET_MISSION_COMMAND(observed))));

guarantee GeofenceMonitor_alert
 "Alert trace property. To be proved by SPLAT" :
 event(alert) <=> (not Policy ->
                (if is_latched then Once (not (Policy)) else not Policy));

guarantee GeofenceMonitor_output
 "The output event fires when alert is not raised" :
 if event(alert) then
   not (event(output))
 else if event(observed) then
   (event(output) and (output = observed))
 else
   not (event(output));
```

Listing 4: The `GeoFence_Monitor` AGREE contract.

AADL connections, memory blocks are shared between sender and receiver components to represent the IPS port state concepts described in Section 3. seL4's *capability* mechanism (configured for port connections using CAmkES can ensure that only the sender/receiver components can access the shared memory and that the information flow is one-way. On other platforms, middleware or underlying OS primitives are used (e.g., for the Linux backend, System V interprocess communication primitives are used).

Semantic consistency across platforms is a fundamental property of HAMR that is achieved by factoring code generation illustrated in Figure 8 through a reference implementation of the AADL RTS described in Section 3. HAMR implements the APIs and platform-independent aspects of the AADL RTS functionality in Slang (described in Section 2). Slang can be compiled to efficient embedded C without incurring garbage collection runtime, as objects are statically allocated (which in turn can be compiled using existing compilers to a wide variety of platforms). Slang's *extension facility* enables Slang programs to interface with full Scala and Java when compiling to the JVM, and C when compiling to C. The ability to provide a common implementation of many aspects of the AADL run-time and application interfaces helps ensure a consistent system implementations across difference HAMR-supported platforms. It also enables both Slang and C to be used to code component application code for HAMR's C-based Linux and seL4 platform backends.

Figure 8 illustrates that the HAMR translation architecture utilizes Slang as much as possible to code platform-independent aspects of the AADL run-time, and then uses Slang extensions in Scala and C to implement platform-dependent aspects. For example, for the JVM platform, a Slang AADL RTS Reference Implementation is used for most of infrastructure implementation, with a few customizations (denoted by the circled "+") written in Scala. For the C-based platforms including seL4, some of the Slang Reference Implementation is inherited, but customizations define memory lay-

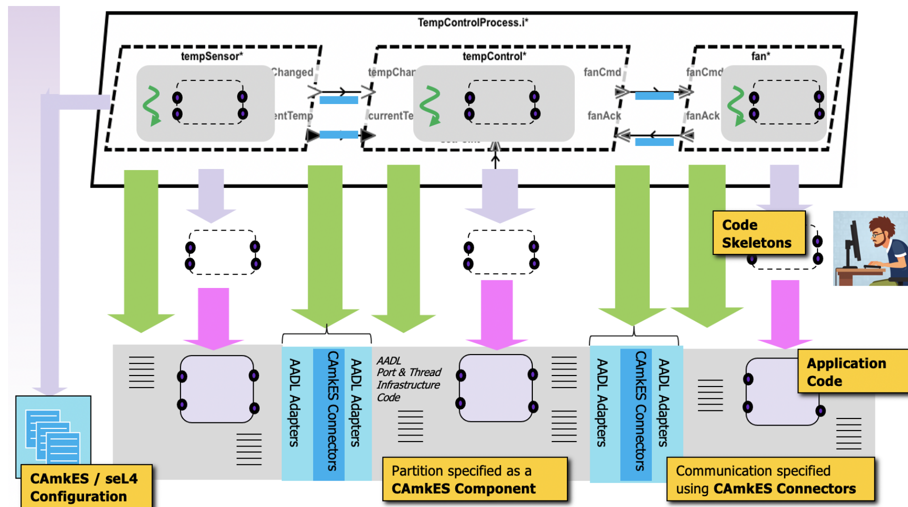

Figure 7: HAMR to seL4 Code Generation Concepts

outs to be used in C (still written in Slang to support eventual verification). Then the Slang-based infrastructure is compiled to C – giving a sizable code base that is shared across Linux and seL4 with some further C customization for each platform.

6. Platform-Independent C Application Logic

As illustrated in Figure 8, code generation for developer-facing APIs for port communication and thread skeletons are all derived from the Slang reference implementation. This means that regardless of platform, the C APIs and skeletons that the developer programs to will be identical; moreover, their abstract structure and typing will be identical to the corresponding Slang. This makes it quite easy to port C-based implementations across HAMR C-based platforms. This is also leveraged to achieve interoperability between HAMR components implemented in different languages (e.g., C and Slang). This section illustrates selected developer-facing skeletons and APIs for the Temperature Control example.

6.1. Auto-Generated Code Skeletons and APIs

For each thread component, HAMR generates skeletons for AADL entry points (see Section 6.4), which the developer will utilize to code the application logic for the thread.

```
// === Initialize Entry Point ===
// illustrate full generated unique name
Unit t_TemperatureControl_TempSensor_i_tsp_tempSensor_initialize_()
{
  // ... fill in application code
}

// === Compute Entry Point (timeTriggered method) ===
// illustrate abbreviated name (for figure formatting)
```

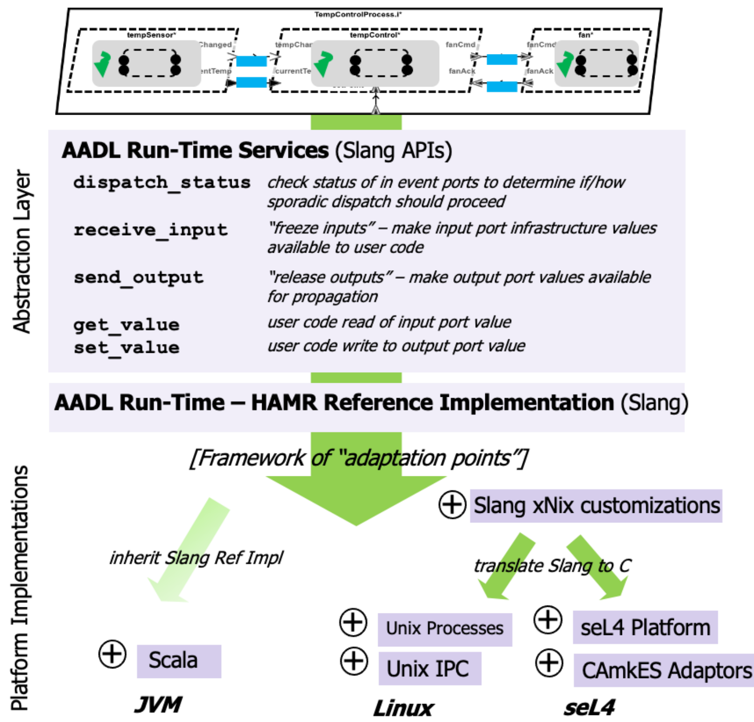

Figure 8: Code Generation Factored Through AADL RTS

```
Unit timeTriggered()
{
 // ... fill in application code for Compute entry point
}
```

Listing 5: TempSensor Entry Point Skeletons

Listing 5 depicts the code skeletons auto-generated for the `TempSensor` thread. The name of the Initialize entry point illustrates that name mangling is used, based on the position of the thread instance within the AADL-specified architecture, to guarantee that the name is unique across the generated code for the entire system. To make the code figures easier to read and format, the names of all other similar methods have been manually simplified.

To facilitate development, HAMR tailors the structure of a thread's Compute entry point according to its `Dispatch_Protocol` thread property specified in the AADL model. Since `TempSensor` has a `Periodic` dispatch protocol, HAMR generates a single method named `timeTriggered` that will be invoked at the start of the thread's period.

```
// === Initialize Entry Point ===
Unit initialize()
{
  // ... fill in application code
}
```

```
// === Compute Entry Point (message handlers) ===
Unit handle_fanAck(FanAck_Type value) {
  // ... fill in application code
}

Unit handle_setPoint(SetPoint_i value) {
  // ... fill in application code
}

Unit handle_tempChanged() {
  // ... fill in application code
}
```

Listing 6: TempControl Message Handlers (excerpts, constituting Compute Entry Point)

Listing 6 shows the `TempControl` entry point skeletons. The structure of the Initialize entry point is the same as that for `TempSensor`. `TempControl` has a `Sporadic` dispatch protocol, and thus it will be dispatched upon the arrival of messages on its input event or event data ports. To tailor the Compute entry point structure to the event-driven character, HAMR generates a message handler method skeleton for each input event and event data port. The Compute entry point application logic is programmed by implementing these handlers. For event data ports, message handlers have a parameter corresponding to the data type declared on the port, e.g., the parameter `value` of type `SetPoint_i` in `handle_setPoint`. The value supplied in the parameter is obtained by the underlying infrastructure using the `GetValue` RTS. For event ports, handlers have no parameters since there are no corresponding payloads on arriving events (i.e., they are simply notifications). HAMR also supports a single entry point method for sporadic components. In that case, the `DispatchStatus` RTS can be called to determine the port who received a message to trigger the dispatch.

For each input port on a component, HAMR generates an API `api_get_<port name>` to access the "frozen" content of the port. Similarly, for each output port, an API `api_put_<port name>` is generated to place a value on the port. These operations are, in effect, versions of the `GetValue` and `PutValue` RTS of Section 3.2 dedicated to specific ports. In the implementations of the `api` methods, the generic `GetValue` and `PutValue` behind the scenes to achieve the functionality of the methods.

```
// auto-generated API for Put Value RTS for currentTemp port
void api_put_currentTemp(Temperature_i value);

// auto-generated API for Put Value RTS for tempChanged port
void api_put_tempChanged();
```

Listing 7: TempSensor Port Communication APIs

Listing 7 shows the `TempSensor` port APIs. For `currentTemp` data port, the parameter `value` is the value (or a reference to the value, depending on the nature of the C type generated from the AADL data model specification). For `tempChanged`, there is no parameter since event port messages are simple signals with no payloads.

```
// auto-generated API for Get Value RTS for currentTemp port
bool api_get_currentTemp(Temperature_i value);

// auto-generated API for Get Value RTS for fanAck port
bool api_get_fanAck(FanAck_Type *value);
```

```
// auto-generated API for Get Value RTS for setPoint port
bool api_get_setPoint(SetPoint_i value);

// auto-generated API for Put Value RTS for fanCmd port
void api_put_fanCmd(FanCmd_Type value);

// auto-generated API for Get Value RTS for currentTemp port
bool api_get_tempChanged();
```

Listing 8: TempControl Port Communication APIs

Listing 8 shows the `TempControl` port APIs. The `put` API for `fanCmd` follows the same structure as the APIs in `TempSensor` component. For the `get` APIs for `currentTemp`, `fanAck`, and `setPoint`, a reference parameter is passed to indicate the storage into which the retrieved value is to be placed (`Temperature_i` and `SetPoint_i` are reference types). For `api_get_tempChanged in event` port, no parameter is passed since there is no payload in the arriving messages. For the event and event data ports, `get` will dequeue the arrived message and it will be not present on subsequent dispatches. For data port `currentTemp`, the arriving value is not dequeued – it will be available to read in subsequent dispatches until overwritten by the arrival of a new value. For the `put` APIs, the supplied values are held in the APS and then released to the communication infrastructure all at once the application code completes (see the concept discussion in Section 3.2 and the example infrastructure code in Section 6.4).

6.2. Auto-generated Data Types

```
// includes defs of AADL user-defined (non Base Type)
// used in current definition
#include <TempUnit_Type.h>

// define abbrev for reference to C struct for this type
typedef struct Temperature_i *Temperature_i;

// C struct representing AADL Data Model struct
struct Temperature_i {
  TYPE type;
  F32 degrees;
  TempUnit_Type unit;
};

// macro for declaring struct values of this type
#define DeclNew_Temperature_i(x)
  struct t_TemperatureControl_Temperature_i x =
     { .type = T_Temperature_i }
```

Listing 9: HAMR Auto-generated Temperature C Type Representation

For each AADL `data` component defined using the AADL Data Modeling framework, HAMR generates a C type representation. Listing 9 shows the auto-generated representation of the AADL `Temperature_i` declaration from Listing 3. A C `struct Temperature_i` is defined along with a `typedef` with the same name for pointers

to the `struct` values.[8] The typedef is used for parameter type declarations like the ones in Listings 7 and 8. The C `struct` includes an additional field `type` to hold a run-time type tag used within the infrastructure code. A macro is defined for declaring variables for `struct` values that automatically initializes the type tag. C representations for all types from the AADL `Base_Types` package are also defined.

6.3. Application Logic

```
// === Component Local Variable
struct Temperature_i lastTemperature;

// === Initialize Entry Point ===
Unit initialize() {
  //  initialize component local variable
  lastTemperature = createTempInFahrenheit(80.0);

  // initialize outgoing data port
  api_put_currentTemp(&lastTemperature);
}

// === Compute Entry Point (timeTriggered method) ===
Unit timeTriggered() {
  // stack-allocate local struct value for temperature
  DeclNew_Temperature_i(currTemp);

  // read current temperature from hardware sensor
  senseTemperature(&currTemp);

  // take action if temperature has changed
  if (lastTemperature.degrees != currTemp.degrees) {
    lastTemperature = currTemp;
    api_put_currentTemp(&lastTemperature);
    api_put_tempChanged();
  }
}
```

Listing 10: TempSensor Application Code

Listing 10 shows the completed application code for the `TempSensor` entry points. The developer has added a component local variable `lastTemperature` whose value will persist between thread dispatches. Following the objectives given in Section 3 for the AADL Initialize entry point, the Initialize entry point allocates an initial struct value to be held in the persistent component local variable `lastTemperature`, and the `put` API is used to initialize the `currentTemp` data port.

On each periodic dispatch, the Compute entry point will stack-allocate storage to hold the most recent temperature via the `DeclNew` macro. Then `senseTemperature` is called to retrieve the current temperature from the temperature sensor. If the current temperature differs from the previous, a new temperature value is put on the `currentTemp` data port and a notification is on the `tempChanged` event port.

```
Unit handle_tempChanged() {
  DeclNew_Temperature_i(currTemp); // allocate new temperature struct
```

[8] Using the same name for the struct and for the typedef is valid because these are associated with separate name spaces in C. The use of overlapping names simplifies our code generation strategy.

```
  if (api_get_currentTemp(&currTemp)) {
    struct Temperature_i currTempInF =
        convertTemperatureToFahrenheit(&currTemp);

    if (currTempInF.degrees > setPoint.high.degrees) {
      api_put_fanCmd(FanCmd_Type_On);
    } else if (currTempInF.degrees < setPoint.low.degrees) {
      api_put_fanCmd(FanCmd_Type_Off);
    }
  }
}
```

Listing 11: TempControl Application Code (tempChanged handler)

Listing 11 shows the completed `tempChanged` event handler for the `TempControl` thread. The `get` port API is used to fetch the `currentTemp`. If the operation is successful, the temperature is compared to the stored latest `setPoint` to determine the appropriate command (if any) to sent to the fan.

6.4. Component Infrastructure

Figures 7 and 8 helped illustrate how HAMR-generated implementations use the AADL RTS to provide a layer of code that realizes the basic steps of the AADL run-time while abstracting away the details of the underlying platform. Furthermore, HAMR implements significant portions of the AADL run-time in Slang, which provides easier inspection and verification and can be translated to all of HAMR's target platforms (insuring semantic consistency).

```
def compute(): Unit = {
 // perform AADL RTS ReceiveInput
 Art.receiveInput(eventInPortIds, dataInPortIds)

 // call Compute entry point application code
 component.timeTriggered(operational_api)

 // perform AADL RTS SendOutput
 Art.sendOutput(eventOutPortIds, dataOutPortIds)
}
```

Listing 12: Generated TempSensor AADL Infrastructure – Slang Reference Implementation for Compute Entry Point wrapper (excerpts)

To illustrate these concepts with a small fragment of the run-time, Listing 12 shows excerpts of the auto-generated Slang code for the component infrastructure of the `TempSensor` thread (this is some of the code from the gray areas of Figure 7 labeled "Port & Thread Infrastructure Code"). In HAMR translation architecture of Figure 8, this code is associated with area labelled "HAMR Reference Implementation (Slang)". The `Compute` method is invoked by the run-time's scheduling framework.[9] Following the concepts of Figure 4, the HAMR `Art` (AADL Run Time library) `ReceiveInput` service is called to move port data from the communication infrastructure into view of the application, thus "freezing" the port values. The application

[9]There are a few subtleties regarding alignment with the AADL standard's presentation of thread *dispatching* and *running*. The standard has distinct steps/states of *dispatching* and *running*. However, the current HAMR implementation unifies these into a single step (this will be changed in the future). Due to our static scheduling approach with seL4, the code in Listing 13 is sufficiently close to the standard's intent.

code for the Compute entry point is invoked (the `timeTriggered` method in this periodic thread corresponding to Listing 10). After the application code has executed, `SendOutput` service is called to move any output port information set by `put` calls to the communication infrastructure.

```
Unit tempSensor_Bridge_EntryPoints_compute_(
 STACK_FRAME tempSensor_Bridge_EntryPoints this) {
  DeclNewStackFrame(
   caller,
   "tempSensor_Bridge.scala",
   "tsp_tempSensor_Bridge.EntryPoints", "compute", 0);

 // perform AADL RTS ReceiveInput
 sfUpdateLoc(93); {
  art_Art_receiveInput(SF (IS_82ABD8)
    tempSensor_Bridge_EntryPoints_eventInPortIds_(this),
    (IS_82ABD8)
    tempSensor_Bridge_EntryPoints_dataInPortIds_(this));
  }

  // call Compute entry point application code (timeTriggered)
  sfUpdateLoc(96); {
    tempSensor_timeTriggered(
     SF (TempSensor_i_Operational_Api)
     tempSensor_Bridge_EntryPoints_operational_api_(this));
  }

  // perform AADL RTS SendOutput
  sfUpdateLoc(98); {
    art_Art_sendOutput(
     SF (IS_82ABD8)
     tempSensor_Bridge_EntryPoints_eventOutPortIds_(this),
     (IS_82ABD8)
     tempSensor_Bridge_EntryPoints_dataOutPortIds_(this));
  }
}
```

Listing 13: Generated TempSensor AADL Infrastructure – Compute Entry Point Wrapper Transpiled from Slang (excerpts)

Listing 13 shows the results of transpiling the Slang code of Listing 12 to C. In HAMR translation architecture of Figure 8, this code is associated with area labelled “translate Slang to C”. This code is still platform-indendent in that it is used for both Linux and seL4 deployments. Instantiations of the functions in the code, in particular the `ReceiveInput` and `SendOutput` will use the seL4 representations for ports and threads discussed in Section 7 (corresponding to the bottom right of Figure 8). The relationship between Listings 12 and 13 is straightforward, but it is important to note that the Slang/C transpiler inserts additional code to support traceability to the Slang code and execution. First, the `sfUpdateLoc` macros (which can be selectively enabled and disabled for debugging) provide information about the originating line numbers (e.g., `93`) in the Slang code. Second, similar macros like `STACK_FRAME` and `SF` are used to maintain stack trace information so that any execution of the C code can be traced back to conceptual execution of the originating Slang code using the stack trace information (which describes the execution stack in terms of Slang artifacts). The build framework can be configured to remove this traceability code in the operational deployment to the platform.

7. HAMR seL4 Backend

This section overviews the HAMR AADL code generation strategy for seL4. Section 7.1 describes how CAmkES is used to configure the kernel and establish the partitioning and communication topology for the AADL system. Section 7.2 details how the semantics of AADL threading, including scheduling, is represented using the seL4 domain scheduler and CAmkES threading. Section 7.3 describes how the semantics of AADL port-based communication is represented using CAmkES ports and connections. For both threading and communication, significant engineering has been involved in generating adapter/infrastructure code from HAMR to bridge the gap between AADL semantics, CAmkES primitives and seL4 platform functionality.

7.1. CAmkES and Kernel Configuration

Recall from Section 3 that, whenever HAMR generates code for seL4, the input AADL models must have each thread component as the only component within a process component. Each AADL process component is mapped to a CAmkES component. seL4 implements each CAmkES component instance as a non-overlapping address space partition, with strict protections enforced by the kernel to maintain spatial isolation between components. In the baseline CAmkES as used in the Trusted Build tool [7], an seL4 thread was generated for each input "port" on component. Since this did not align with AADL's notion of threading, the CAmkES translation was modifed to have a single seL4 thread within a CAmkES component. Thus, the final representation of an AADL process/thread combination within seL4 is aligned with structure of HAMR seL4 AADL models: an AADL process represents a protected address space (achieved via a CAmkES component and its translation to seL4) and there is a single thread executing with the process.

```
1  component TempSensor_i_tsp_tempSensor {
2    ...
3    emits ReceiveEvent sb_tempChanged;
4    emits TickTock sb_self_pacer_tick;
5    consumes TickTock sb_self_pacer_tock;
6    dataport sp_union_art_DataContent_t sb_currentTemp;
7    dataport sb_event_counter_t sb_tempChanged_counter;
8  }
9
10 component TempControl_i_tcp_tempControl {
11   ...
12   emits ReceiveEvent sb_fanCmd_1_notification;
13   consumes ReceiveEvent sb_fanAck_notification;
14   consumes ReceiveEvent sb_tempChanged;
15   dataport sp_union_art_DataContent_t sb_currentTemp;
16   dataport sb_queue_union_art_DataContent_1_t sb_fanAck_queue;
17   dataport sb_queue_union_art_DataContent_1_t sb_fanCmd_queue_1;
18   dataport sb_event_counter_t sb_tempChanged_counter;
19   has semaphore sb_dispatch_sem;
20 }
21
22 component Fan_i_fp_fan {
23   ...
24   emits ReceiveEvent sb_fanAck_1_notification;
25   consumes ReceiveEvent sb_fanCmd_notification;
26   dataport sb_queue_union_art_DataContent_1_t sb_fanCmd_queue;
27   dataport sb_queue_union_art_DataContent_1_t sb_fanAck_queue_1;
28   has semaphore sb_dispatch_sem;
29 }
```

Listing 14: Excerpts from the Generated CAmkES Representation of the Temperature Control Components

Listing 14 gives excerpts of the CAmkES component declaration for each of the processes/threads in the temperature control system of Section 3 (see the corresponding AADL `TempControl` specification in Listing 1). Each AADL port is represented by a pair of CAmkES ports (e.g., one for notification of arrival of data, one for storage of data) as described in Section 7.3. Additional CAmkES elements such as the `TickTock` ports and `semaphores` are added as described in Section 7.2 to support the HAMR/seL4 scheduling regime.

```
1  assembly {
2    composition {
3      component TempSensor_i_tsp_tempSensor tsp_tempSensor;
4      component TempControl_i_tcp_tempControl tcp_tempControl;
5      component Fan_i_fp_fan fp_fan;
6
7      connection seL4SharedData
8        conn1(from tsp_tempSensor.sb_currentTemp,
9              to tcp_tempControl.sb_currentTemp);
10     connection seL4Notification
11       conn2(from tsp_tempSensor.sb_tempChanged,
12             to tcp_tempControl.sb_tempChanged);
13     connection seL4SharedData
14       conn3(from tsp_tempSensor.sb_tempChanged_counter,
15             to tcp_tempControl.sb_tempChanged_counter);
16     connection seL4Notification
17       conn4(from tcp_tempControl.sb_fanCmd_1_notification,
18             to fp_fan.sb_fanCmd_notification);
19     connection seL4SharedData
20       conn5(from tcp_tempControl.sb_fanCmd_queue_1,
21             to fp_fan.sb_fanCmd_queue);
22     connection seL4Notification
23       conn6(from fp_fan.sb_fanAck_1_notification,
24             to tcp_tempControl.sb_fanAck_notification);
25     connection seL4SharedData
26       conn7(from fp_fan.sb_fanAck_queue_1,
27             to tcp_tempControl.sb_fanAck_queue);
28     connection seL4Notification
29       conn8(from tsp_tempSensor.sb_self_pacer_tick,
30             to tsp_tempSensor.sb_self_pacer_tock);
31   }
32
33   configuration {
34     tsp_tempSensor._stack_size = 110592;
35     tsp_tempSensor._domain = 2;
36     tcp_tempControl._stack_size = 110592;
37     tcp_tempControl._domain = 3;
38     fp_fan._stack_size = 110592;
39     fp_fan._domain = 4;
40     tsp_tempSensor.sb_currentTemp_access = "W";
41     tcp_tempControl.sb_currentTemp_access = "R";
42     tsp_tempSensor.sb_tempChanged_counter_access = "W";
43     tcp_tempControl.sb_tempChanged_counter_access = "R";
44     tcp_tempControl.sb_fanCmd_queue_1_access = "W";
45     fp_fan.sb_fanCmd_queue_access = "R";
46     fp_fan.sb_fanAck_queue_1_access = "W";
47     tcp_tempControl.sb_fanAck_queue_access = "R";
48   }
49 }
```

Listing 15: Excerpts from the Generated CAmkES Representation of the Temperature Control System

Walking over the structure in the AADL instance model, HAMR generates a CAmkES `assembly` as shown in Listing 15 to define the partitioning and communication topology of the system deployment on seL4. In the `configuration` section, the `_access` properties are used to define the component read/write access to the shared-memory-based representations of the AADL. These declarations provide part of the foundational realization of information flow control within the system and they are guaranteed to be enforced by formally verified seL4 kernel. The `_domain` properties specify the scheduling domain for each component/partition and are used to achieve temporal separation of the AADL processes/threads following the scheme described in Section 7.2.

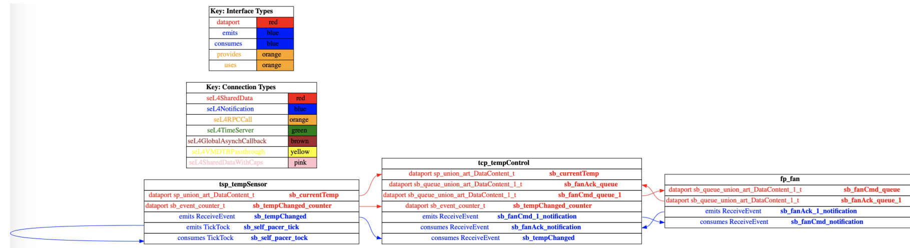

Figure 9: CAmkES Topology Diagram Generated by HAMR

To support traceability and understanding of the seL4-level deployment, HAMR generates a diagram of the CAmkES topology as shown in Figure 9.

7.2. Representing AADL Threads in seL4

To satisfy high-assurance execution requirements suitable for, e.g., mission-critical avionics, including real-time performance and non-interference, HAMR generates configurations to enforce scheduling requirements and temporal isolation between threads. HAMR accomplishes this by extracting the thread topology and properties as specified in the AADL model. The topology provides communications dependencies which will drive schedule generation. Properties include real-time performance requirements, e.g., rates and execution durations of threads, and maximum allowed communications latencies. Next, HAMR aligns the AADL thread activation and dispatching with CAmkES notions of activation/yielding in CAmkES and seL4 threading primitives by inserting wrapper code (organized into AADL entry points) into CAmkES components to host the application level code, as presented in the previous section.

Temporal isolation prevents an errant or compromised thread from interfering with processing cycles of other execution threads. Since many real-time control systems require multiple threads to function correctly, simple priority schemes that only guarantee performance of the highest level threads are inadequate. Additionally, while these simple priority schemes may satisfy basic properties, such as number-of-cycles-per-frame, they do not necessarily satisfy sub-frame latency and jitter requirements. Depending on the system-level requirements, this may lead to instabilities, as well as provide undesirable signaling opportunities to exfiltrate information.

Therefore, HAMR supports a statically scheduled AADL and runtime model that guarantees that all threads receive their required (as specified in the AADL model) execution resources, at times that satisfy their real-time requirements (again, as specified

in the AADL model), such as rate and required duration of processing cycles for execution on the underlying platform. HAMR generates infrastructure code to leverage an seL4 feature called the *domain scheduler*. This feature is currently available in the formally verified version of seL4. The domain scheduler is a mechanism accessible via CAmkES that allows the system designer to specify a static cyclic periodic schedule for the resident components. Each CAmkES component is assigned to a non-overlapping temporal domain (e.g., the `_domain` property shown on lines 35 and 39 in Listing 15) Background threads maintained by the kernel, such as threads for managing communication memory and other kernel operations, are by default assigned to Domain Zero, although they can be reassigned to support mission-level requirements. The static cyclic domain schedule is then comprised of an ordered sequence of execution intervals, each interval assigned a single temporal domain and an internal length (defined in terms of number of processor ticks). During runtime, the seL4 kernel strictly enforces the domain schedule defined for the system, iterating through the intervals, giving each interval its defined time, and then returning to the first interval when the end is reached. A domain schedule may not be altered or disabled during run-time.

To provide determinism, mission systems, such as avionics, are dispatched, read inputs, perform their processing, write outputs, and wait until the next period. The domain scheduler by itself, however, lacks a "start of frame" signal to awaken threads. Threads therefore have no intrinsic way to determine where they should be in their execution sequence.[10] One approach is to rely on inputs from upstream components in order to start. This is supported by seL4 and CAmkES. However, if a compromised or failed upstream component stops sending those signals, all downstream processing would halt. Another approach is to have each thread continuously sample its inputs. Such *busy-wait* approaches consume power and resources that could be better applied elsewhere, including partition-specific background processing.

To address this gap, we developed a signaling approach to synchronize the components to the periodic frame. In this context, the periodic frame (or major frame in terminology common to real-time scheduling) is the cumulative length of time in which each of the components executes a single iteration. The synchronization mechanism "wakes up" the components at specific intervals. HAMR implements the synchronization through a custom CAmkES component called the *pacer*, as shown at the left in Figure 10. The pacer component emits a single event notification at the frame interval to each of the other components in the system. Each component then waits on this notification signal before starting its regular iterative execution.

The pacer works by leveraging several features of the underlying seL4 domain-based approach: as mentioned above, threads for managing communication memory and other kernel operations, are by default assigned to Domain Zero. We can schedule when Domain Zero is active. We place the pacer in its own domain. When that pacer domain is active, and the pacer initializes, it *emits* a "tick" notification to itself. The

[10]The Mixed Criticality System (MCS) version of seL4 provides this capability. At the time of this writing, MCS is not yet fully verified. Due to the CASE emphasis of presenting a vision for formal-methods-based assurance to the broader DoD community, a decision was made to stick with the verified version of the kernel.

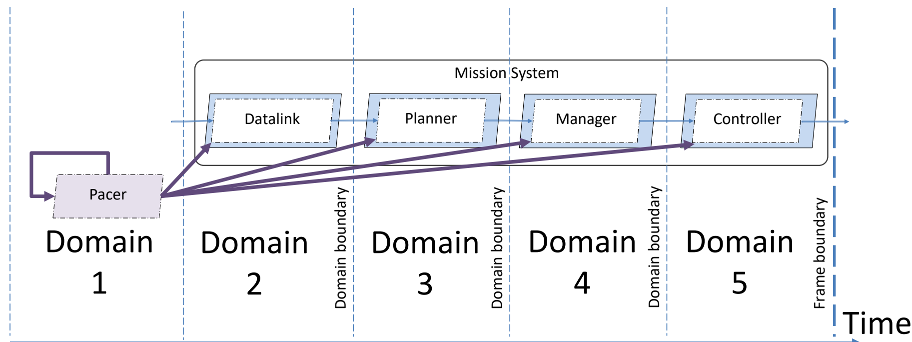

Figure 10: The pacing component provides a start-of-frame signal to threads executing under the seL4 domain scheduler.

pacer then blocks, waiting for that notification on its "tock" input. This is shown in Figure 11 That notification will not appear until Domain Zero next activates. When the pacer's domain is again active, the pacer will be released from its wait. The pacer will then emit start-of-frame notifications to all the other threads in their domains, as well as a tick-tock signal to re-awake itself. The sequence then repeats.

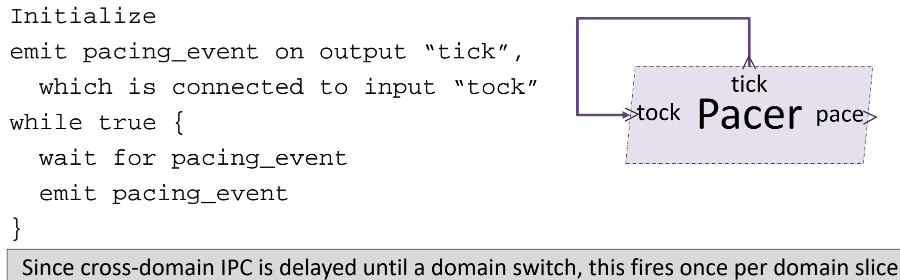

Figure 11: Pacer component implementation signals itself with a tick-tock signal to activate once per frame.

Since the pacer is an underlying infrastructure element, the pacer component does not need to be captured in the AADL model. Instead, HAMR auto-generates a single pacer component in the system, including its communication channels to each of the other components. HAMR also auto-generates a wrapper around the application components that implements behavior code to wait on the pacer signal. This allows the system designer to drop in their behavior code for the components into the system build via a stable API without writing special code to handle the pacer events.

An explicit pacer is not necessary in all cases; we can also generate self-pacing components using the same feature that the pacer component itself uses for pacing. For example, the `TempSensor` component in Listing 14 uses self-pacing; the relevant pacing ports are shown on lines 4 and 5. Line 28-30 of Listing 15 declares the wrap-around connection. For sporadic components, such as the `TempControl` and `Fan` components in Listing 14, HAMR generates semaphores ports for the thread dis-

patches on lines 19 and 29. Pacing a virtual machine or other components that have long durations split across multiple activations may require additional specialized pacer components. Multi-rate schedules can be implemented using a pacer per rate.

Within the HAMR infrastructure, the domain schedule for a system is captured in a file named `domain_schedule.c`. A snippet of a domain schedule example is shown below. If explicit pacers are used, it too must have time explicitly allocated in the domain schedule.

```
1  const dschedule_t ksDomSchedule[] = {
2    { .domain = 0, .length = 100 }, // all other threads
3    { .domain = 2, .length = 5 },   // tempSensor
4    { .domain = 3, .length = 5 },   // tempControl
5    { .domain = 4, .length = 5 },   // fan
6    { .domain = 0, .length = 380 }, // pad rest of period
7  };
8  const word_t ksDomScheduleLength =
9    sizeof(ksDomSchedule) / sizeof(dschedule_t);
```

Listing 16: An Excerpt of the Generated Domain Schedule for the Temperature Control System

This schedule was generated by HAMR using AADL property annotations, such as the one shown in the model excerpt below. This AADL example specifies that the `TempSensor` should be assigned to domain 2 (`CASE_Scheduling_Domain`) and allowed to execute at most 10 microseconds when scheduled (`Compute_Execution_Time`). The length of time between clock ticks is specified by the `Clock_Period` property (which in this case targets ODROID-XU4). Thus the length of the domain iteration for the `TempSensor` is set to 5 ticks.

```
1  processor implementation TempControlProcessor.i
2    properties
3      Frame_Period => 1000ms;
4      Clock_Period => 2ms;
5      ...
6
7  thread TempSensor extends BasicThread
8    ...
9    properties
10     Dispatch_Protocol => Periodic;
11     Period => 1000ms;
12     CASE_Scheduling::Domain => 2;
13     Compute_Execution_Time => 10ms .. 10ms;
```

Listing 17: Timing Property Annotations in a Sample AADL Model

The developer may customize the schedule once it has been generated, for example to assign a different domain execution order. Furthermore, the system designer is responsible for engineering a domain schedule that satisfies the timing requirements of the components. If one or more of the components are not given enough time to execute properly, then the overall system may fail or produce incorrect results. If the components are given too much time to execute, then the system operates inefficiently. Existing AADL-based analysis tools can evaluate a model annotated with timing properties and determine if a periodic schedule can satisfy the specified temporal requirements of all the components, including latency, jitter, and execution durations [30].

7.3. Representing AADL Port Communication in seL4

To satisfy high-assurance communications requirements suitable for domains such as mission-critical avionics, including both determinism and separation, HAMR enforces strict one-way communication, as well as communications availability and integrity. It accomplishes this by extracting the system requirements as specified in the AADL model, and interpreting those requirements as per the well-defined AADL runtime semantics.

One-way communications flow helps stakeholders (e.g., developers, maintainers, certifiers) understand how information flows through the system. Without the ability to specify strict one-way flows, back-channels may allow components to interact in ways not intended by the developers. For example, the baseline CAmkES did not provide one-way communication with existing primitives; it merely supported read-write of memory locations by the sender, and read-only by the receiver. However, since control and data-flow concepts were mixed, the receiver could, at a minimum, signal the sender.

At the AADL level, ports have explicit directionality, and the model-level analysis tools check that the specified connectivity is satisfied by the interconnection network. Using underlying communications mechanisms that violate the specified connectivity would mean that there would be no way to guarantee that the implementation matched the requirements as specified by the model. Therefore, analysis at the model level would have to be fully repeated at the implementation level to determine that information flow requirements were satisfied. Earlier attempts to provide one-way communications flows (e.g. the baseline Trusted Build tool [7]) relied on intermediate components and Remote Procedure Call (RPC) mechanisms, which required additional threads, and suffered back channels and blocking. It enforced read-write of shared memory locations by the sender, and read-only by the receiver. However, control and data-flow concepts were mixed, and the receiver could, at a minimum, signal the sender.

Potentially more significant, the particular RPC mechanisms used could permit a receiver catch that does its own work and blocks the sender from getting work done. For example, performing `seL4_Call` on an Endpoint requires the Sender to trust that the receiver will reply in order to unblock it or requires the sender implement its own timeout mechanism that would require another supervising thread reset the blocked thread when the receiver doesn't reply. `Call` blocks until the message is transferred to the receiver and keeps blocking until the receiver replies. `seL4_Send` blocks until message is transferred to receiver, and `seL4_NBSend` will transfer message if receiver is waiting and not transfer message if receiver is not waiting. All these mechanisms offer undesirable signalling capabilities that can violate strict partitioning requirements. This means that the sender had to trust that the receiver RPC mechanism was implemented correctly, otherwise it could lead to blocking the sender. This would violate criticality partitioning boundaries; the receiver would have to be verified to the same criticality as the sender. Below we describe how we solved these issues.

Our first step is to have HAMR enforce that only one-way flows may be specified in the AADL model. If a particular communication channel requires a return acknowledgment from the receiving component to the sending component, then the acknowledgment notification must be explicitly modeled in AADL. This modeling exposes those communications channels to analysis tools, so insecure channels that violate partitioning boundaries may be identified early in the design process.

Communications availability and integrity is the property that a correctly configured run-time satisfies the rate, jitter, and latency requirements of the communications as specified and verified in the model. Integrity includes the property that dropouts are detectable (e.g., if the sender drops out, the receiver should be able to perceive the inputs are stale, or if the receiver drops, when it recovers, it should be able to perceive that it missed inputs). In addition, the receiver must be able to detect incomplete messages (e.g., the sender was unable to finish sending the message before the receiver read the message, or the receiver was interrupted before completing the read, and a new message arrived in the meantime). AADL specifies buffering semantics, so mission system developers can balance communications buffering to satisfy system-level, mission-specific requirements. HAMR extracts these channel-specific properties from the AADL model, and generates corresponding infrastructure code.

To provide the seL4 platform-specific representations for infrastructure port state (IPS) described in Section 3.2, HAMR generates shared memory buffers for port communications, as shown in Figure 12. HAMR automatically configures the read/write privileges on this shared memory to match the requirements as derived from each connection in the AADL model. For example, again consider the prior excerpts of the generated CAmkES in Listing 14 for the `TempControl` and `Fan` components along with the portion of the CAmkES assembly that connects their `fanCmd` ports in Listing 15. Access restrictions are introduced on lines 40-47 in the configuration section to ensure, for example, a producer can only write to its shared data (access type "W") and a consumer can only read from it (access type "R").

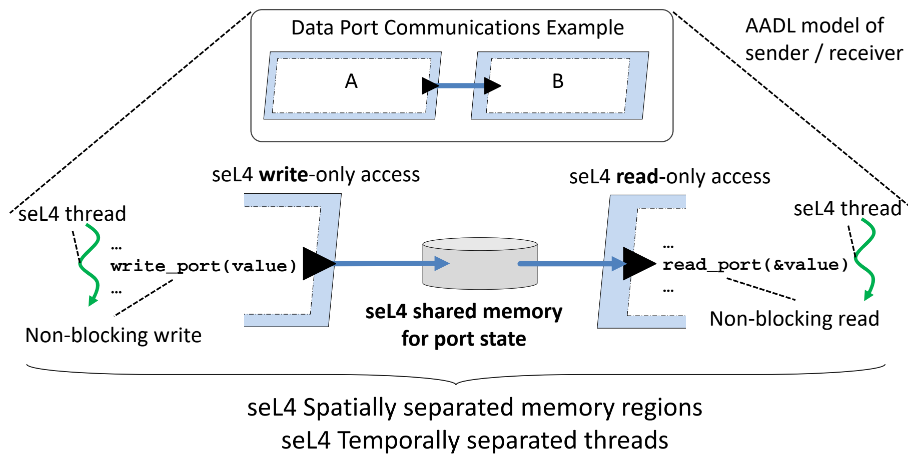

Figure 12: Shared memory with temporal and spatial controls provides non-blocking communications to implement AADL semantics suitable for mission-critical communications

At the infrastructure level, HAMR enforces the AADL-specific semantics for AADL event ports, data ports, and data event ports with custom writer/readers. The event data port dequeue code is shown in Listing 18. This can be generalized to also handle pure event ports (no data, just an event counter) and data ports (queue length 2).

```c
bool queue_dequeue(recv_queue_t *recvQueue, counter_t *numDropped, data_t *data) {
  counter_t *numRecv = &recvQueue->numRecv;
  queue_t *queue = recvQueue->queue;
  // Get a copy of numSent so we can see if it changes during read
  counter_t numSent = queue->numSent;
  // Acquire memory fence - ensure read of queue->numSent BEFORE reading data
  __atomic_thread_fence(__ATOMIC_ACQUIRE);
  // How many new elements have been sent? Since we are using unsigned
  // integers, this correctly computes the value as counters wrap.
  counter_t numNew = numSent - *numRecv;
  if (0 == numNew) {
    // Queue is empty
    return false;
  }
  // One element in the ring buffer is always considered dirty. It's the next
  // element sender will write.  It's not safe to read it until numSent has been
  // incremented. Thus there are really only (QUEUE_SIZE - 1) elements in the
  // queue.
  *numDropped = (numNew <= QUEUE_SIZE - 1) ? 0 : numNew - QUEUE_SIZE + 1;
  // Increment numRecv by *numDropped plus one for the element we are about to
  // read.
  *numRecv += *numDropped + 1;
  counter_t numRemaining = numSent - *numRecv;
  size_t i = (*numRecv - 1) % QUEUE_SIZE;
  *data = queue->elt[i]; // Copy data
  // Acquire memory fence - ensure data read BEFORE reading queue->numSent again
  __atomic_thread_fence(__ATOMIC_ACQUIRE);
  if (queue->numSent - *numRecv + 1 < QUEUE_SIZE) {
    // Sender did not write element we were reading. Copied data is coherent.
    return true;
  } else {
    // Sender may have written element we were reading. Copied data may be
    // incoherent. We dropped the element we were trying to read, so
    // increment *numDropped.
    ++(*numDropped);
    return false;
  }
}
```

Listing 18: Example seL4 event data port dequeue

This approach supports multiple readers, with no side channels between readers. It does not work for multiple writers, due to potential side-channels between writers. We therefore prohibit multiple writers from being specified in the AADL models.

There are potentially detectable micro-architectural interactions at the individual memory cell level. This depends on both processor and compiler level optimizations and is outside the scope of what can be covered at the pure software level.

HAMR automatically maps the various AADL ports to CAmkES `dataport DataContent` and `Counter` ports, such as shown earlier in Listing 14, lines 6-7, 15-18, and 26-27. HAMR then generates adapters to convert the CAmkES dataport APIs to AADL compliant APIs, such as shown in Listing 19.

```c
void TempControl_tempChanged_Rcv(Opt_DataContent result) {
  if(tempChanged_dequeue()) {
    DeclEmpty(payload);
    DeclSome_DataContent(some);
    Some_DataContent_init(&some, (DataContent) &payload);
    assign(result, &some, sizeof(union Opt_DataContent));
  } else {
    DeclNone_DataContent(none);
    assign(result, &none, sizeof(union Opt_DataContent));
  }
}
```

Listing 19: Example seL4 glue code

8. Evaluation

This section reports on several publicly available model/code bases used to evaluate HAMR. The HAMR distribution is available at [33]. Each example repository includes a ReadMe file with graphical and textual summaries of the example, statistics about the code base, and instructions on how to run the example. The purpose of the examples is to illustrate different aspects of HAMR code generation, APIs, and system execution.

8.1. Temperature Control – JVM, Javascript, Linux, and seL4 Platforms

The Temperature Control example is the basic "Hello World" example used to illustrate HAMR workflows for Slang/JVM, Javascript, Linux, and seL4 deployments, HAMR application code APIs, and a variety of other HAMR support including a unit testing framework, guided simulation, and system visualizations. The full set of artifacts for the Temperature Control example presented in this paper are available in a public GitHub repository [36]. The AADL model includes 3 thread components (1 periodic, 2 sporadic), 9 thread component ports, and 4 connections between thread ports. The size of the component application code (code associated with thread entry points) is 74 NSLOC. The size of the HAMR-generated code together with the application code is 17402 NSLOC. This represents the size of the system before CAmkES is run, which generates a substantial amount of kernel-level code for the executable deployment image.

The repository also includes a HAMR-generated Linux deployment. A Slang-based version is available at [34]. The seL4 code base is the subject of a two-hour video tutorial (slides and videos available on the HAMR website [33] by Feb 1, 2022).

8.2. Isolette: Infant Incubator Controller – JVM, Javascript, Linux, and seL4 Platforms

The Isolette example illustrates HAMR's ability to take a single Slang-based implemention of application logic of a system and deploy the system on four different platforms (Slang/JVM, Javascript, Linux, and seL4). In contrast to the Temperature Control example described in this paper, the application code for the Isolette example is written in Slang and is then transpiled to C. The full set of artifacts is available in a public GitHub repository [34]. The Isolette example is taken from the US Federal Aviation Administration (FAA) Requirements Engineering Management Handbook (REMH), where it is used to illustrate best practices in requirements engineering for critical embedded systems. An Isolette is an infant incubator (medical device), and the REMH presentation focuses on the heat (infant warming) control subsystem and the safety monitoring subsystem. The REMH includes detailed requirements at multiple levels of abstraction and some design aspects.

An AADL model was constructed from this information, and HAMR was used to develop an implementation of the two subsystems in Slang. The architecture (directed by the REMH description) emphasizes periodic threads and data ports. The

control system and the safety monitoring system include three periodic threads each. An additional periodic thread is used to implement/simulate the operator interface. Slang extensions were used to simulate the temperature sensor and heater components. HAMR generates the JVM deployment of the system. Scala and Java are used to develop the simulated hardware elements and the GUI for the operator interface. In the JVM deployment, there are 9 thread components, 47 thread component ports, and 25 connections between thread ports, with 6349 NCSLOC in the infrastructure code and 507 NCSLOC in the application logic. For the Linux and seL4 deployments, the Slang to C transpiler is used to compile Slang component implementations and Slang-based platform-independent aspects of the AADL run-time. The resulting C code is composed with the platform-specific aspects coded in C as well as C-based simulations of the hardware elements. For Linux, the combined translated C has 37938 NCSLOC. For seL4, the resulting HAMR-generated C is of similar size, and to that is added the C code generated from CAmkES.

8.3. UAV System – seL4 Platform

This example from the DARPA CASE program summarized in Section 4 demonstrates HAMR's ability to handle system elements representative of those found in a complex high-assurance aerospace system with cyber-resiliency requirements. In addition to the features exercised in the examples above, this example includes virtual machine components, integration with application code written in CakeML, as well as significantly larger application data structures passed via port-based communication between components. The full set of artifacts is available in a public GitHub repository [35]. The AADL model includes 9 thread components (including one component representing a virtual machine). The size of the component application code (code associated with thread entry points) is 806 NSLOC (this excludes the metrics for the attestation gate, geofence monitor, and the line search task filter as their behavior is supplied by CakeML code). The size of the HAMR-generated code together with the application code (both C code and CakeML code) is 42235 NSLOC. This represents the size of the system before CAmkES is run, which generates a substantial amount of kernel-level code for the executable deployment image.

This is example is used in CASE-related training material for Collins Aerospace and Lockheed Martin CASE teams.

9. Related Work

Trusted Build: HAMR can be seen as a successor to the Trusted Build (TB) tool [7] developed in the DARPA High Assurance Cyber Military Systems (HACMS) Program by Collins Aerospace, University of Minnesota, and the seL4 team. Like HAMR, TB generated component skeletons for seL4 from AADL using the CAmkES seL4 component modeling language. TB was the first AADL translation framework to seL4, and it was used in DARPA HACMS to construct several systems at roughly the same complexity as the UAV system described in Section 8.3. TB was instrumental in demonstrating the vision for model-based development for seL4, and many of the team members that provided support and guidance for TB on DARPA HACMS have continued to play similar roles with HAMR on DARPA CASE.

An earlier version of HAMR seL4 backend used the same CAmkES and C code structure as TB, but evolved to provide a number of additional capabilities beyond TB. At the CAmkES level, the port-based inter-component communication strategy was upgraded to provide true one-way communication from the sender to the receiver on an AADL connection as described in Section 7. In the TB strategy, it was possible to have some backflow of information, which is undesirable from an information assurance perspective. The CAmkES patterns employed in TB had unnecessary complexity that made it harder in general to support information assurance arguments. In addition, the TB port-based communication structure introduced an extra monitoring component for each AADL connection – dramatically increasing the number of CAmkES components and associated support threads of the generated system and making it much more difficult to realize the more capable scheduling approach now used in HAMR/seL4. The TB generated structures also did not support AADL semantics for ports employing AADL's richer dispatching strategies (e.g., port urgency, explicit indication of ports that trigger dispatch) and port value freezing (as described in Section 3). TB provided no support for automated VM creation (engineers needed to manually configure VMs in an empty generated CAmkES component shell), whereas the HAMR VM support significantly reduces engineer effort as well as the potential for defects. HAMR also adds enhanced support for QEMU-based emulation and dramatically reduces the effort needed to create a working development environment by using a Vagrant setup framework.

Ocarina: Ocarina, led by Hugues [19], is the longest running AADL code generation project. Written in Ada and supported by a plug-in to OSATE, Ocarina provides backends for Ada and C code generation primarily using PolyORB-HI [31]. PolyORB-HI is a lightweight middleware designed for high-integrity systems. Ocarina generates real-time tasking and communication infrastructure for C-based RT-POSIX threading, the Xenomai framework that provides real-time support on top of Linux, and the open source RTOS RTEMS. The PolyORB-HI Ada implementation is used with the GNAT compiler to support full Ada on native platforms (e.g., Linux, Windows) and the Ravenscar Ada subset profile to guarantee schedulability and safety properties. Ocarina also has a backend for POK, a partitioned operating system compliant with the ARINC 653 standard, along with configuration file generation for ARINC 653-compliant DeOS and VxWorks653 real-time operating systems (RTOS).

Ocarina has been used in several European defense industry projects over the last 12-15 years. Whereas the industry focus for Ocarina has primarily been for RTOSs, we have focused HAMR on the seL4 microkernel for cyber-resiliency and information assurance. While Ocarina and HAMR both support multiple backends, Ocarina emphasizes targeting the common structure of the C and Ada PolyORB-HI implementations, while HAMR emphasizes factoring backends through language-independent standardized run-time services. AADL RTS is currently supported, but the system is modular so others can be supported.

Ocarina currently focuses on integrating code generation for RTOSs with integrated schedulability analysis. HAMR currently has an industrial research focus to move from the JVM-based framework for prototyping, visualization, and coding in a clean modern language subset (Slang) that can be compiled to C and from there to industry platform

deployments. HAMR's current industrial research projects (e.g., DARPA CASE) have prioritized the use of the machine verified seL4 microkernel. HAMR is being used in conjunction with Adventium Labs FASTAR AADL temporal analysis and schedule-generation tools.

RAMSES: The code generation approach of the Refinement of AADL Models for Synthesis of Embedded Systems (RAMSES) tool [4] emphasizes successive automated AADL model refinement. The refinement steps are driven by developer-specified features for the target system, by capabilities and resources of the target platform, and by model-level analyses that assess system properties against requirements and platform capabilities. Such analyses include schedulability, timing properties, and resource analysis. By gradually exposing more implementation details in the model, those details can be considered in the analysis. The incremental transformations also form the basis of a correctness methodology in which the correctness of each transformation is considered. Once model transformations yield a sufficiently detailed implementation model, RAMSES generates C component infrastructure, that when combined with developer-written component application C code, can be deployed on Linux (with POSIX-compliant threading), nxtOSEK (open-source platform for LEGO Mindstorms), and POK. RAMSES has been used to develop systems for the avionics, railway, and robotics domains.

The differences in emphasis between HAMR's target application areas and RAMSES roughly correspond to the HAMR/Ocarina differences detailed above. In addition, HAMR supports multiple languages and distinct platforms. RAMSES emphasizes model transformations as a basis for correctness arguments whereas Ocarina and HAMR emphasize factoring through abstract architecture layers. Like Ocarina, RAMSES focuses more on RTOS applications compared to HAMR's current focus on microkernel-based information assurance and multi-platform support. Compared to HAMR, one challenge of the RAMSES approach is that the refinement steps produce multiple versions of AADL models. Multiple versions require additional work to maintain traceability and correspondence between the model-level contracts and information flow requirements and the source-code level contracts.

10. Conclusion

Industry experience across a number of domains is increasingly demonstrating the effectiveness of formally verified microkernels as a foundation for high-assurance systems. We believe that the overall effectiveness of verified microkernels such as seL4 can be improved, and adoption can be accelerated, by providing modeling and development environments with abstractions that are aligned with kernel configurations and with system engineering needs of the domain. AADL is a strong candidate for exploring model-driven development for microkernels. The HAMR framework described in this paper demonstrates that the component-oriented idioms of AADL can provide effective abstractions for developers to configure the partitioning and information control mechanisms provided by seL4. AADL semantics for threading and port-based communication are based on decades of experience in the embedded domain; experience using AADL to develop the mission control software systems in the DARPA CASE

effort confirms the appropriateness of AADL semantics. Combining the capabilities of AADL modeling with seL4 kernel controls via HAMR code generation yields a powerful synergy that enables the forward-looking forms of model-based development exemplified by the BriefCASE tool set. In particular, when using a platform-based approach and modeling environment that is designed from "top to bottom" to emphasize component-wise development with strong semantics for both components and infrastructure, engineers can more easily specify, analyze, transform, assure, and evolve their systems.

The approach that we have illustrated is not limited to AADL or seL4. Since HAMR is designed to facilitate the integration of new translation backends, and as other AADL code generation tools have demonstrated support for other real-time operating systems, we are confident that additional HAMR backends can be added that will enable the entire BriefCASE vision to be realized more broadly. On the modeling language front, the important aspect of our approach is adherence to AADL's run-time semantics for threading, communication, and infrastructure services. Other modeling languages, including SysML variants that have a notion of component or blocks and connections, could potentially be used as modeling front ends.

Looking ahead, providing support for seL4 Mixed-Criticality Scheduling is a high priority. For this, we are prototyping a new scheduling framework that is aligned with the AADL approach to scheduling that can be "dropped into" the seL4 MCS infrastructure. We are also investigating HAMR extensions to support distributed systems based on middleware frameworks like OMG's Data Distribution Service (DDS) used in US military systems and other domains. On the assurance front, we are continuing to enhance HAMR's support for evidence generation to feed into assurance case construction. We are documenting assurance case templates and illustrating these with assurance arguments for the BriefCASE tool chain. Within the HAMR code base itself, we are applying the Logika Slang verification framework to verify key functionality.

11. Acknowledgments

This work was funded in part by DARPA contract HR00111890001, as well as by the US Air Force Research Laboratory and the US Army. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *1st International Conference Principles of Security and Trust (POST)*, volume 7215 of *LNCS*, pages 369–389, 2012.

[2] T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve. *Specification and Checking of Software Contracts for Conditional Information Flow (extended version)*, pages 341–379. Springer, 2010.

[3] I. Amundson and D. Cofer. Resolute assurance arguments for cyber assured systems engineering. In *Design Automation for Cyber-Physical Systems and Internet of Things (DESTION 2021)*, May 2021.

[4] E. Borde, S. Rahmoun, F. Cadoret, L. Pautet, F. Singhoff, and P. Dissaux. Architecture models refinement for fine grain timing analysis of embedded systems. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 44–50, 2014.

[5] A. Burns and A. Wellings. *Analysable Real-Time Systems: Programmed in Ada*. CreateSpace, 2016.

[6] D. Cofer, I. Amundson, J. Babar, D. Hardin, K. Slind, P. Alexander, J. Hatcliff, Robby, G. Klein, C. Lewis, E. Mercer, and J. Shackleton. Cyber assured systems engineering at scale. 2021. Submitted to *IEEE Security & Privacy*.

[7] D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart. A formal approach to constructing secure air vehicle software. *Computer.org*, 51:14–23, 2018.

[8] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In A. E. Goodloe and S. Person, editors, *NASA Formal Methods*, pages 126–140, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[9] J. Delange. *AADL In Practice: Become an expert in software architecture modeling and analysis*. Reblochon Development Company, 2017.

[10] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2013.

[11] D. S. Hardin and K. L. Slind. Formal synthesis of filter components for use in security-enhancing architectural transformations. In *Proceedings of the Seventh Workshop on Language-Theoretic Security (LangSec 2021), 42nd IEEE Symposium on Security and Privacy*, May 2021.

[12] J. Hatcliff, J. Belt, Robby, and T. Carpenter. HAMR: an AADL multi-platform code generation toolset. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*, volume 13036 of *Lecture Notes in Computer Science*, pages 274–295. Springer, 2021.

[13] J. Hatcliff, J. Hugues, D. Stewart, and L. Wrage. Formalization of the AADL Run-Time Services (extended version). Technical Report 2022-01, SAnToS Laboratory, 2022.

[14] J. Hugues. A correct-by-construction aadl runtime for the ravenscar profile using spark2014. *Journal of Systems Architecture*, 123, Feb. 2022.

[15] D. B. Kingston, S. Rasmussen, and L. R. Humphrey. Automated UAV tasks for search and surveillance. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*, pages 1–8. IEEE, 2016.

[16] F. Kordon, J. Hugues, A. Canals, and A. Dohet. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*. John Wiley & Sons, Inc., 2013.

[17] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.

[18] R. Laddaga, P. Robertson, H. E. Shrobe, D. Cerys, P. Manghwani, and P. Meijer. Deriving cyber-security requirements for cyber physical systems. *CoRR*, abs/1901.01867, 2019.

[19] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. Ocarina: An environment for AADL models analysis and automatic code generation for high integrity applications. In F. Kordon and Y. Kermarrec, editors, *Reliable Software Technologies – Ada-Europe 2009*, pages 237–250, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] E. Mercer, K. Slind, I. Amundson, D. Cofer, J. Babar, and D. Hardin. Synthesizing verified components for cyber assured systems engineering. In *24th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021)*, October 2021.

[21] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[22] National Information Assurance Partnership. Separation Kernel Protection Profile, 2011. `https://www.niap-ccevs.org/Profile/Info.cfm?PPID=65&id=65`.

[23] T. Patten, D. Mitchell, and C. Call. Cyber attack grammars for risk-cost analysis. In *Proceedings of the 15th International Conference on Cyber Warfare and Security*, Norfolk, VA, 2020.

[24] Robby and J. Hatcliff. Slang: The Sireum programming language. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*, volume 13036 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2021.

[25] J. Rushby. The design and verification of secure systems. In *8th ACM Symposium on Operating Systems Principles*, volume 15(5), pages 12–21, 1981.

[26] SAE International. SAE AS5506/2. AADL Annex Volume 2, 2011.

[27] SAE International. SAE AS5506 Rev. C Architecture Analysis and Design Language (AADL), 2017.

[28] H. Thiagarajan, J. Hatcliff, J. Belt, and Robby. Bakar Alir: Supporting developers in construction of information flow contracts in SPARK. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 132–137, 2012.

[29] H. Thiagarajan, J. Hatcliff, and Robby. Awas: AADL information flow and error propagation analysis framework. *Innovations in Systems and Software Engineering (ISSE)*, 2021.

[30] S. Vestal, A. Gordon, K. Schleisman, T. Smith, and R. Whillock. Resource Loading and Timing Analysis Using FASTAR. `https://www.adventiumlabs.com/sites/default/files/documents/FASTAR%20ACVIP%20Training%20Briefing.pdf`.

[31] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008)*, pages 221–228, 2008.

[32] Sireum Awas website. `https://awas.sireum.org`.

[33] HAMR project website. `http://hamr.sireum.org`.

[34] HAMR ISOLA 2021 example artifacts. `https://github.com/santoslab/isola21-case-studies`.

[35] HAMR DARPA CASE phase ii UAV example artifacts (sel4 platform). `https://github.com/santoslab/case-uav-phase2`.

[36] HAMR temperature control example artifacts (sel4 platform). `https://github.com/santoslab/aeic2022_temperature_control`.