HAMR: An AADL Multi-Platform Code Generation Toolset*

John Hatcliff[1], Jason Belt[1] and Robby[1], and Todd Carpenter[2]

[1] Kansas State University, Manhattan KS 66506, USA
{hatcliff,belt,robby}@ksu.edu
[2] Adventium Labs, Minneapolis, MN 55401, USA
Todd.Carpenter@adventiumlabs.com

Abstract. This paper describes the High-Assurance Model-based Rapid engineering for embedded systems (HAMR) tool-kit that generates high-assurance software from standards-based system architecture models for embedded cyber-physical systems. HAMR's computational model is based on standardized run-time services and communication models that together provide an abstract platform-independent realization which can be instantiated by back-end translations for different platforms. HAMR currently targets multiple platforms, including rapid prototyping targets such as Java Virtual Machines, Linux, as well as the formally verified seL4 space partitioned micro-kernel.
HAMR bridges the gap between architecture models and the system implementation by generating high assurance infrastructure components that satisfy the requirements specified in the model and preserving rigorous execution semantics. Based on the architecture model, including the components, their interfaces, run-time performance properties, and inter-component connections, the HAMR-generated code creates Application Programming Interfaces that provide developer-centric ease-of-use, as well as support automated verification.
HAMR currently interprets architecture models captured in the Architecture Analysis and Design Language (AADL). AADL is a rigorous standardized modeling language that has proven useful in the development of high assurance embedded systems. We describe using HAMR for building applications from safety and security-critical domains such as medical devices and avionics mission-systems.

1 Introduction

Advances in model-based engineering (MBE) have improved the development of Cyber-Physical Systems (CPS). A 2009 NASA study documented that nearly 80% of CPS's capability is implemented using software [15], and the trend is increasing. As system and software complexity increases, software integration risk has become a key limiting factor in the development of complex CPSs. A

* Work supported in part by the US DARPA, US Air Force Research Lab, US Army, and the Software Engineering Institute.

study by the SAVI initiative determined that while 70% of errors are introduced at the design phase, most are not found and fixed until integration and test [2]. The study identified that fixing those errors late in the process costs orders of magnitude more than if they had been fixed earlier. This directly impacts system capability due to requirements being cut to keep on schedule or other systems not being built to address budget overruns. In their 2009 study, NASA recommended that the best way to address this is to focus on the system architecture at the design phase, both for new systems as well as system upgrades.

The SAE-standard Architecture, Analysis, and Design Language (AADL) [8] is an system architecture modeling and analysis approach that has obtained a fair amount of traction in the research and aerospace communities. For example, on the System Architecture Virtual Integration (SAVI) effort [3], aircraft manufacturers together with subcontractors used AADL to define a precise system architecture using an "integrate then build" design approach. Working with AADL models, important interactions are specified, interfaces are designed, and integration is verified before components are implemented in code. Once the integration strategy and mechanism are established, subcontractors provide implementations that comply with the architecture requirements. Prime contractors then integrate these components into a system. The integration effort, particularly schedule and risk uncertainty, is reduced due to the previous model-based planning and verification. Due in part to the NASA and SAVI studies, since 2012 the US Army has been investing in developing, maturing, and testing MBE and other engineering capabilities for the system development of the Future Vertical Lift (FVL), a top Army priority to modernize the vertical lift fleet.

One of the challenges inherent with modeling and analysis is maintaining consistency between the model-as-analyzed and the system-as-implemented. Any deviation can lead to inaccuracies in the predictions provided by the models, which can impact system performance against requirements. An example of this is if an implementation decision violates system partitioning requirements.

Another challenge is that many modeling approaches are design-time documentation exercises that are rapidly outdated as the system is implemented, deployed, and maintained throughout the life-cycle. If the models are merely documentation and not maintained and understood by developers, any system updates (e.g., bug fixes, new features) might violate the system-level concepts and requirements.

Yet another challenge is that some generic modeling approaches permit different interpretations of information contained in the models. One example of this is when modelers capture significant model information in comments. This makes it difficult to make system-level decisions based on the integration of components provided by different vendors.

To address these issues, we have developed a tightly integrated modeling and programming paradigm, called HAMR, to shift development to earlier in the design cycle, and thereby eliminate issues earlier, when they are less expensive to address. To accomplish this, HAMR encodes the system-level execution semantics, as specified in standardized models with clear and unambiguous specifications, into infrastructure code suitable for the given target platform. These execution semantics include component interfaces, threading semantics, inter-component communication semantics, standard system phases, scheduling, and application behavioral and non-functional performance properties.

Benefits of this approach include:

Extensible code-generation architecture capable of easily targeting new platforms: Complex industrial systems often include multiple platforms (e.g., in system-of-system architectures), and long-lived systems often need to be migrated to new architectures to support technology refresh.

Support for incremental development, from rapid prototyping to full deployment: Organizations can perform rapid prototyping and integration in spirals, moving in successive spirals (with different code generation back-ends) from simpler functional mockups of component behaviors and interface interactions to more realistic implementations on test bench boards, to final deployments on platform hardware.

Standard Development Environments HAMR runs on widely-used platforms, leveraging development environments that are already familiar to students, graduate researchers, and entry-level industrial engineers. This helps reduce workforce training costs.

Direct support for formally-proven partitioning architectures: Industry teams are increasingly using micro-kernels, separation kernels, and virtualization architectures to isolate critical system components. Strong isolation provides a foundation for building safe and secure systems. It also enables incorporation of legacy components into a system (e.g., running legacy code on virtual machines within a partition of a micro-kernel). HAMR includes a back-end to directly target the seL4 micro-kernel whose implementation, including spatial partitioning, is formally proven correct using theorem-proving technology.

This paper describes HAMR, with specific contributions including:

- HAMR provides code generation for the SAE AS5506 Standard AADL. The code generated by HAMR conforms to the AADL AADL standard's Run-Time Services, and further refines it towards a more precise semantics for safety-critical embedded systems.
- HAMR leverages Slang, a safety/security-critical subset of Scala. We define a Slang-based reference implementation of the above AADL Run-Time Services.
- Using the Slang AADL RTS, we define a Slang/Scala-based AADL code-generation, component development approach, and JVM run-time execution environment that can be used for JVM-based AADL system deployments or system simulations before further refinement to an embedded (e.g., C-based) deployment.
- We define a multi-platform translation architecture code AADL code generation by using the Slang AADL RTS reference implementation as an abstraction layer through which multiple back-ends can be supported.
- We implement C-based back-ends for the translation architecture targeting Linux OS and for the seL4 micro-kernel [13].
- We validate the translation framework using industrial-scale examples from multiple CPS domains including medical devices and military mission control systems.

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

2 AADL

SAE International standard AS5506C [10] 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 precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements. Software components include data, subprogram, subprogram group, thread, thread group, and process. Hardware components include processor, virtual processor, memory, bus, virtual bus, and device. Devices are used to model sensors, actuators, or custom hardware. An AADL `system` component represents an assembly of interacting application software and execution platform components. Each component category has a different, well-defined standard interpretation when processed by AADL model analyses. Each category also has a distinct set of standardized properties associated with it that can be used to configure the specific component's semantics.

A *feature* specifies how a component interfaces with other components in the system. *ports* are features that can be classified as an *event port* (e.g., to model interrupt signals or other notification-oriented messages without payloads), a *data port* (e.g. modeling shared memory between components or distributed memory services where an update to a distributed memory cell is automatically propagated to other components that declare access to the cell), or an *event data port* (e.g., to model asynchronous messages with payloads, such as in publish-subscribe frameworks). Inputs to event and event data ports are buffered. The buffer sizes and overflow policies can be configured per port using standardizes AADL properties. Inputs to data ports are not buffered; newly arriving data overwrites the previous value.

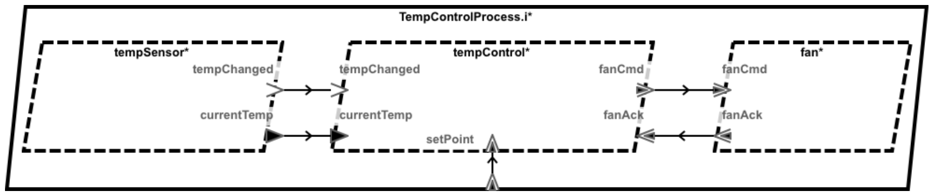

Fig. 1. Temperature Control Example (excerpts) – AADL Graphical View

Figure 1 presents a portion of the AADL standard graphical view for a simple temperature controller that maintains a temperature according to a set point

[3] Source code and supporting documentation available at `https://github.com/santoslab/isola21-hamr-case-studies`

structure containing high and low bounds for the target temperature. 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 read the value on its `currentTemp` *data port* and compare it the most recent set points. If the current temperature exceeds the high set point, it will send `FanCmd.On` `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. AADL editors such as the Eclipse-based Open Source AADL Tool Environment (OSATE) synchronize 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` configure the tasking semantics of the thread.

```
thread TempControl
features
 currentTemp: in data port TempSensor::Temperature.i;
 tempChanged: in event port;
 fanAck: in event data port CoolingFan::FanAck;
 setPoint: in event data port SetPoint.i;
 fanCmd: out event data port CoolingFan::FanCmd;
properties
 Dispatch_Protocol => Sporadic;
 Period => .5 sec;  -- the min sep between incoming msgs
end TempControl;

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

The bottom of the listing declares an implementation named `TempControl.i` of the `TempControl` component type. Typically, when using HAMR, AADL thread component implementations such as `TempControl.i` 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, and further details about the thread's implementation will be found in the source code, not the model). Using information in the associated thread type, HAMR code generation will generate platform-independent infrastructure, thread code skeletons, and port APIs specific for the thread, and a developer codes 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 listing below illustrates how architectural hierarchy is realized as an integration of subcomponents. The body of `TempControlProcess` type has no declared features because the component does not interact with its context in this simplified example. However, the body of the implementation has subcomponents (named component instances), and the subcomponents are "integrated" by declaring connections between subcomponent ports.

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

process implementation TempControlProcess.i
subcomponents
 tempSensor : thread TempSensor::TempSensor.i;
 fan : thread CoolingFan::Fan.i;
 tempControl: thread TempControl.i;
 operatorInterface: thread OperatorInterface.i;
connections
c1:port tempSensor.currentTemp -> tempControl.currentTemp;
c2:port tempSensor.tempChanged -> tempControl.tempChanged;
c3:port tempControl.fanCmd -> fan.fanCmd;
c4:port fan.fanAck -> tempControl.fanAck;
end TempControlProcess.i;
```

AADL editors check for type compatibility between connected ports. HAMR supports data types declared using AADL's standardized Data Model Annex [1]. For example, the data type declarations associated with the temperature data structure are illustrated below.

```
data Temperature
properties
 Data_Model::Data_Representation => Struct;
end Temperature;

data implementation Temperature.i
subcomponents
 degrees: data Base_Types::Float_32;
 unit: data TempUnit;

data TempUnit
properties
Data_Model::Data_Representation => Enum;
Data_Model::Enumerators=>("Fahrenheit","Celsius","Kelvin");
end TempUnit;
```

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 type library. The `unit` field has an application-defined enumerated type.

AADL omits concepts associated with requirements capture and user interactions such as UML use cases, sequence diagrams, as well as class-oriented software units that are more appropriate when modeling general purpose object-oriented software. AADL is closer in spirit to SysML, although AADL elements are more precisely defined to enable analyzeability and tool interoperability. In industry applications of AADL, SysML may be used earlier in the development process to initially capture interactions between and the system and environment as well as rough architecture. AADL is then used to more precisely specify architecture and to support architecture analysis. Though having a workflow with multiple modeling languages is not ideal, the SysML + AADL approach utilizes the capabilities currently available to industry engineers that want to use AADL. In the broader vision of "programming: what's next?", AADL seems to be tracking the right course by more deeply integrating programming and modeling, but there is even more opportunity to integrate, in a single modeling framework, early design concepts that have both stronger semantics and traceability to eventually developed code-level artifacts.

AADL provides many standard properties, and allows definition of new properties. Examples of standard properties include thread properties (e.g., dispatch protocols such as periodic, aperiodic, sporadic, etc., and various properties regarding scheduling), communication properties (e.g., queuing policies on particular ports, communication latencies between components, rates on periodic communication, etc.), memory properties (e.g., sizes of queues and shared memory, latencies on memory access, etc.). User-specified property sets enable one to define labels for implementation choices available on underlying platforms (e.g., choice of middleware realization of communication channels, configuration of middleware policies, etc.).

The Eclipse-based OSATE tool provides an environment for editing AADL and has a plug-in mechanism that supports different AADL analysis tools. Controls for HAMR code generation are implemented as an OSATE plug-in.

3 Architecture

Since it is a code-generation framework, HAMR focuses on AADL software components – especially thread components and port-based communication between threads. The HAMR code-generation backend includes libraries for threading and communication infrastructure that help realize the semantics of AADL on the target platform.

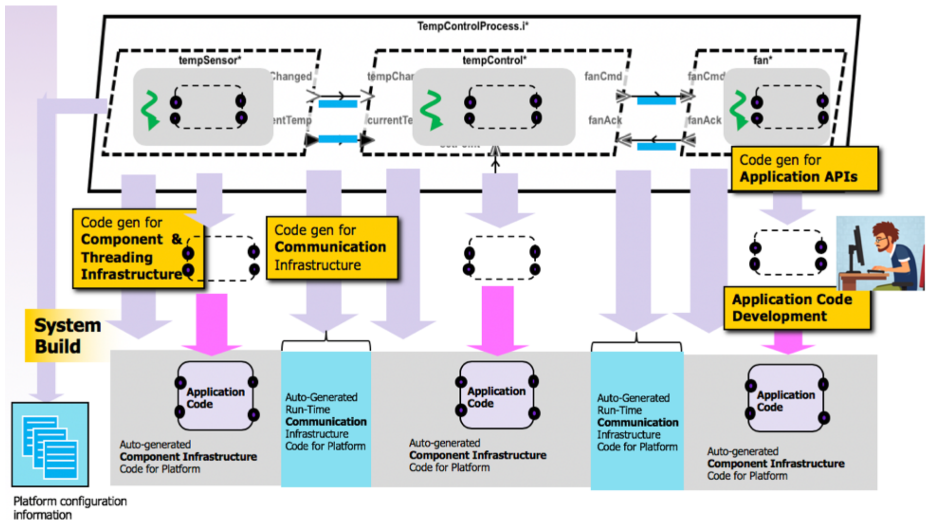
TempControlProcess.i*
tempSensor*
tempControl*
fan*
Changed
entTemp
tempChar
currentTe
fanCmd
fanAck
Code gen for Application APIs
Code gen for Component & Threading Infrastructure
Code gen for Communication Infrastructure
Application Code Development
System Build
Application Code
Auto-generated Component Infrastructure Code for Platform
Auto-Generated Run-Time Communication Infrastructure Code for Platform
Platform configuration information

Fig. 2. HAMR Code Generation Concepts

Figure 2 illustrates the main concepts of HAMR code generation. For each thread component, HAMR generates code that provides an execution context for a real-time task. This includes: (a) *infrastructure code* for linking application code to the platform's underlying scheduling framework, for implementing the storage associated with ports, and for realizing the buffering and notification semantics associated with event and event data ports, 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. For each port connection, HAMR generates infrastructure code for the communication pathway between the source and target ports. On platforms such as seL4, pathways may utilize memory blocks shared between the components (seL4's *capability* mechanism can ensure that only the source/destination 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 Linux, HAMR uses System V interprocess communication primitives.

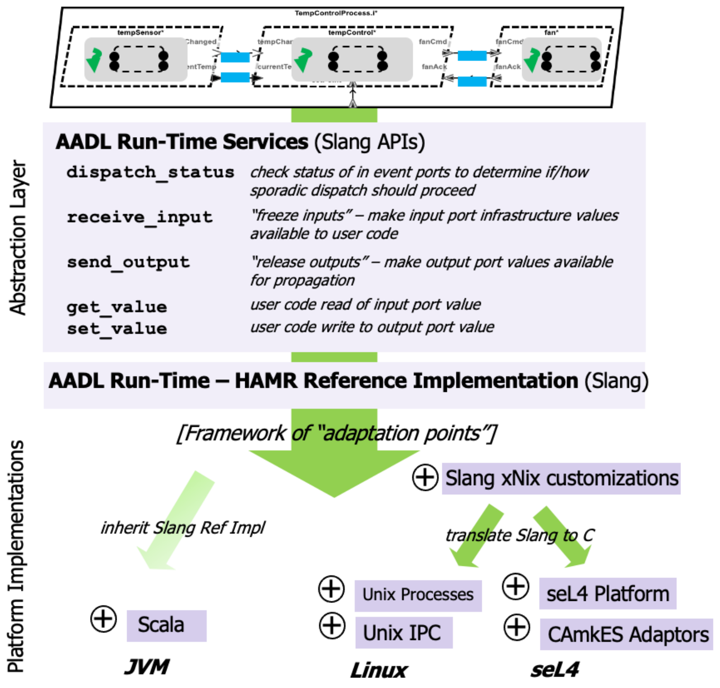
TempControlProcess.i*
tempSensor*
Changed
entTemp
tempChan
currentTe
tempControl*
fanCmd
fanAck
fan*
fanCmd
fanAck
AADL Run-Time Services (Slang APIs)
dispatch_status
check status of in event ports to determine if/how sporadic dispatch should proceed
receive_input
"freeze inputs" – make input port infrastructure values available to user code
send_output
"release outputs" – make output port values available for propagation
get_value
user code read of input port value
set_value
user code write to output port value
Abstraction Layer
AADL Run-Time – HAMR Reference Implementation (Slang)
[Framework of "adaptation points"]
Slang xNix customizations
inherit Slang Ref Impl
translate Slang to C
Unix Processes
seL4 Platform
Scala
Unix IPC
CAmkES Adaptors
JVM
Linux
seL4
Platform Implementations

Fig. 3. Code Generation Factored Through AADL RTS

Semantic consistency across platforms – that is, identical behavior of HAMR-generated code regardless of the target platform – is a fundamental HAMR goal. Semantic consistency is supported by carrying out the code-generation in stages. In particular, code is generated first for a platform-independent reference implementation of the AADL run-time services (RTS) (run-time libraries providing key aspects of threading and communication behavior) as illustrated in Figure 3. These services are currently described informally in the AADL standard via textual descriptions of APIs for thread dispatching and port communication. HAMR specifies the APIs and platform-independent aspects of the AADL RTS functionality in Slang – a subset of Scala designed for high-assurance embedded system development. The HAMR-provided realization of these services is a "reference implementation" in the sense that (a) the highly-readable Slang APIs and service implementations can be directly traced to descriptions in the AADL standard and (b) the subsequent implementations on different platforms are derived from these Slang artifacts. For example, Slang can be compiled to Java Virtual Machine (JVM) bytecode and to efficient embedded C without incurring runtime garbage collection 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.

Figure 3 illustrates that the HAMR translation architecture utilizes Slang 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 (xNix) platforms, some of the Slang Reference Implementation is inherited but customizations define memory layouts to be used in C (still written in Slang to support eventual verification). Then the Slang-based infrastructure is compiled to C. This provides a sizable code base that is shared across Linux and seL4 with some further C customization for each platform.

The Slang-to-C compilation also enables developers to code component application logic in Slang when targeting the JVM or C-based platforms (including Linux and seL4 described in this paper) or C alone for C-based platforms. While this architecture does not currently include formal proofs of conformance of the generated code to Slang reference implementation and associated semantics, it is architected to prepare for such assurance in future work.

4 HAMR Backends

In this section, we describe three HAMR backend targets: (1) JVM, (2) Linux (native), and (3) seL4. The JVM target is provided to quickly implement component and system functionality on a widely-available platform that can be easily utilized without having to set up a RTOS target. This is effective for teaching AADL model-based development principles, and the HAMR JVM platform architecture is set up to eventually support distributed and cloud-based applications via industry-standard publish-subscribe frameworks like JMS, DDS, and MQTT. In two ongoing US DoD funded projects, a contract-based verification framework is being developed that supports integrated AADL and Slang-level contracts with automated SMT-based verification support. In a recently completed industrial project milestone, the JVM platform was used by industry engineers to quickly mock-up and test component functionality and simulate the overall system behavior, including being able to test specific component schedule orderings. If Slang is used to implement component behaviors, such implementations can also be compiled to C along with the Slang-based AADL RTS middleware that HAMR generates specific for the system. The overall system can then be run natively on Linux (as well as on macOS and Windows/Cygwin, with some environment setups). By leveraging Slang extension language facility, developers can opt to implement component behaviors partly/fully in C, for example, to leverage existing libraries, to access hardware, or to integrate legacy components. The seL4 verified micro-kernel backend supports on embedded system boards such as ODROID-C2/XU4. HAMR also generates deployments for seL4 on the QEMU simulation framework, which can be used for testing before deploying to actual hardware. As part of the DARPA CASE project, we provide a Vagrant file to automatically provision a VirtualBox Linux Virtual Machine (VM) with HAMR and its dependencies configured, including OSATE, compiler toolchains, and seL4; it is available at [18]. We continually update the Vagrant definition as we refine HAMR, as well as integrating new or enhanced seL4 features. HAMR has early support for additional targets such as FreeRTos, Minix3, and STM32, but these are less mature than the above.

Below we introduce the key concepts of code architecture using Slang, and then subsequent sections (relying on code examples in the appendix) illustrate the C-based coding.

4.1 Slang on JVM Platform

An AADL thread with sporadic dispatch mode is dispatched upon the arrival of messages on its input event or event data ports. To tailor the application code structure of a sporadic thread Compute Entry Point to the event-driven character, HAMR generates a message handler method skeleton for each input event and event data port. To program the application logic of the component, the developer completes the implementation of these method handlers.
For example, for the sporadic `TempControl` thread, HAMR generates the following Slang skeletons for entry points (excerpts).

```
@msig trait TempControl_i {
  // reference to APIs to support port communication
  def api : TempControl_i_Bridge.Api

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

  // == Skeletons for Compute Entry Point ==
  // handler for the ‘tempChanged‘ input event port
  def handletempChanged(): Unit = {
    // auto-generated default implementation simply logs
    //   messages
    api.logInfo("received tempChanged")
    api.logInfo("default tempChanged implementation")
  }

  // handler for the ‘fanAck‘ input event data port
  def handlefanAck(value:TempControl.FanAck.Type):Unit={
     api.logInfo(s"received ${value}")
     api.logInfo("default fanAck implementation")
  }

  // handler for the ‘setPoint‘ input event data port is
  // similar to above and omitted.

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

To complete the Initialise Entry Point, the developer codes any initialization of component local variables that persist between activations of the thread, e.g., the variable caching the most recent set point structure is initialized. The initial values for all output data ports must also be set (not applicable in this case, because the component has no output data ports), and optionally, messages may be sent on output event and event data ports. The Finalise Entry Point is also coded to perform any clean up steps (omitted).

```
var setPoint: SetPoint_i = Defs.initialSetPoint

override def initialise(): Unit = {
  // The Initialize Entry Point must initialize all
  // component local state and all output data ports.

  // initialize component local state
  setPoint = Defs.initialSetPoint

  // initialize output data ports
  //  (no output data ports to initialize)
}
```

The primary application logic of the `TempControl` is coded by filling in the auto-generated skeletons for the message handler methods. The completed handler for `tempChange` port is given below. The code illustrates the use of auto-generated `api` methods to send and receive information on ports. These provide a uniform abstraction of the AADL communication semantics and allows HAMR to generate different implementations when deploying on different platforms.

```
override def handletempChanged(): Unit = {
  // get current temp from currentTemp data port
  // using auto-generated APIs for AADL RTS
  val temp = api.getcurrentTemp().get
  // convert current temp to Fahrenheit
  val tempInF = Util.toFahrenheit(temp)
  // convert stored setpoint values to Fahrenheit
  val setPointLowInF = Util.toFahrenheit(setPoint.low)
  val setPointHighInF = Util.toFahrenheit(setPoint.high)

  val cmdOpt: Option[FanCmd.Type] =
    if (tempInF.degrees > setPointHighInF.degrees)
      Some(FanCmd.On)
    else if (tempInF.degrees < setPointLowInF.degrees)
      Some(FanCmd.Off)
    // if current temp is between low and high set point
    // don't produce a command (None)
    else None[FanCmd.Type]()

  cmdOpt match {
    // if a command was produced, send it
    // using auto-generated API for AADL RTS
    case Some(cmd) =>
      api.sendfanCmd(cmd)
    case _ =>
      // temperature OK
  }
}
```

`TempSensor` is a periodic thread, and so instead of generating event handlers for the Compute Entry Point, HAMR generates a single `TimeTriggered` method.

```
object TempSensor_i_p_tempSensor {

def initialise(api:TempSensor_i_Initialization_Api):Unit={
    // initialize outgoing data port
    val temp = TempSensorNative.currentTempGet()
    api.setcurrentTemp(temp)
  }

def timeTriggered(api:TempSensor_i_Operational_Api):Unit={
    val temp = TempSensorNative.currentTempGet()
    api.setcurrentTemp(temp)
    api.sendtempChanged()
  }
}
// extension interface to step outside the Slang
// language subset
@ext object TempSensorNative {
  def currentTempGet(): Temperature_i = $
}
```

This example illustrates the use of the Slang *extension* mechanism that is used to interface to code outside of the Slang language subset. On the JVM platform, this typically involves interfacing to classes in full Scala or Java, e.g., to implement GUIs, simulation of sensors and actuators, or JNI interfaces to access GPIO facilities on a development board. In this example, a Slang extension interface is declared to pull a temperature value from the sensor. Multiple implementations of an extension interface may be set up to switch between, e.g., a simulated sensor and interfacing to an actual hardware sensor. The listing below illustrates a simple Scala-implemented sensor simulation that generates randomized values directed by the current state of extension simulation for the Fan hardware.

```
object TempSensorNative_Ext {
  var lastTemperature
           = Temperature_i(68f, TempUnit.Fahrenheit)
  var rand = new java.util.Random

  def currentTempGet(): Temperature_i = {
    lastTemperature = if (rand.nextBoolean()) {
      val delta =
        F32((rand.nextGaussian() * 3).abs.min(2).toFloat
          * (if (FanNative_Ext.isOn) -1 else 1))
      lastTemperature(degrees
         = lastTemperature.degrees + delta)
    } else lastTemperature
    return lastTemperature
  }
}
```

Corresponding to the gray areas of Figure 2, HAMR generates code for each component infrastructure that links the developer-code application logic above to the threading/scheduling mechanisms of the underlying platform. The listing illustrates the pattern of an auto-generated Compute Entry Point for a sporadic thread, which processes messages on incoming event/event-data ports (using the AADL RTS (`Art`) `dispatchStatus` and `receiveInput`) and then calls corresponding developer-written message handlers. After handlers complete, the AADL RTS `sendOutput` is called to propagate data on output ports to connected consumers. During an execution, the `compute` method of each thread is called by an executive that follows a selected scheduling strategy.

This code illustrates some fundamental properties of the AADL computational model – namely, its input/work/output structure of task activations. First, similar to other real-time models designed for analyzeability (e.g., [5]), a task's interactions with its context are cleanly factored into inputs and outputs. At each task activation, inputs are “frozen” for the duration of the task’s activity (which runs to complete within a WCET bound). The AADL RTS `receiveInput` freezes input by moving values from the communication infrastructure into the user thread’s space (dequeuing event and event data port entries as necessary). During the task work, user code can read the frozen values using the `getXXX` APIs and can prepare outputs using `setXXX`. After the task’s work is completed (e.g., event handler completes), the prepared outputs are released to the infrastucture using the `sendOutput` RTS.[4]

```
def compute(): Unit = {
    // get ids of ports that have pending messages.
    val EventTriggered(portIds)
             = Art.dispatchStatus(TempControl_i_BridgeId)
    // "freeze" data ports -- move data port values
    //         from infrastructure to application space
    Art.receiveInput(portIds, dataInPortIds)
    // --- invoking application code (event handlers) ---
    // for each arrived event, call corresponding handler
    for (portId <- portIds) {
      // if an event arrived on the ‘fanAck‘ port
      if (portId == fanAck_Id){
        // get payload, call fanAck handler
        //  with the message payload as parameter
        val Some(BuildingControl.FanAck_Payload(value))
                = Art.getValue(fanAck_Id)
        component.handlefanAck(value)
      } else if(portId == setPoint_Id){
        val Some(BuildingControl.SetPoint_Payload(value))
                = Art.getValue(setPoint_Id)
        component.handlesetPoint(value)
      } else if(portId == tempChanged_Id) {
       // ‘tempChanged‘ port is event (not event data)
       //     so there is no payload to pass to handler
         component.handletempChanged()
      }
    }
    // after all handlers run, propagate to consumers
    //   the values that they wrote to output ports
   Art.sendOutput(eventOutPortIds, dataOutPortIds)
 }
```

The overall system is run by launching a HAMR-generated JVM application of the system. Once launched, the application infrastructure initializes the AADL RTS middleware (e.g., allocate objects representing communication channels) calls the `initialize` entry point of each component. Then the executive infrastructure is called which repeatly invokes `compute` methods according to the scheduling strategy. During a shut down phase, each component’s `finalise` entry point is called.

HAMR auto-generates unit test harnesses for each component with helper methods for loading values into input ports, invoking the various entry points, and checking values on output ports. Also included is a run-time monitoring framework where, e.g., all send/receive actions on ports are logged on a Redis server which can be filtered in a variety of ways using a HAMR-generated framework that utilizes Akka and ReactiveX stream processing and filtering.

[4] The `compute` code shown above deviates from the AADL standard description slightly in that the `for` loop processes one queued message on each incoming event port. An alternate implementation aligned with the standard is available that only processes a single event and then releases its output and yields.

This framework is used to generate multiple visualizations of the system execution (including dynamic generation of message sequence charts reflecting inter-component communication.

4.2 Linux

HAMR supports Linux natively by translating the Slang-based AADL RTS implementations to C. Slang has a memory model that enables memory to be statically allocated when translated to C, and it supports highly-controlled data type representations and other constructs that enable effective embedded code to be generated. The high-level infrastructure APIs, coordinating procedures of the Slang-based AADL RTS reference implementation, and Slang-based platform customizations for Linux constitute the infrastructure code that is translated to C. Using the Slang extension mechanism, only around 100 SLOC of C code are linked into the infrastructure to provide the lowest level aspects of the inter-component communication using Unix System V shared memory interprocess communication. Everything else is written in Slang, which lays the groundwork for future formal verification of the infrastructure code and makes it easy to establish traceability to the Slang-based AADL RTS reference model.

In the current organization of the generated infrastructure code, each thread component runs as a separate Linux process due to industrial project emphasis on separation. In the upcoming phases of projects, we will be investigating alternate approaches that allow multiple AADL threads to be grouped in the same AADL and Linux process. For implementing component application logic, two different workflows are supported: (1) a thread's behavior can be coded in Slang as in the previous section and compiled to C, or (2) C-level entry point APIs can be generated and coded/debugged in a C development environment.

4.3 seL4 Verified Microkernel

One of the goals of DARPA CASE program is help DoD industry teams harden systems to make them more resilient to cyber-attacks. The seL4 micro-kernel (formally verified using automated theorem-proving techniques) is a central part of the CASE approach. seL4 provides a *capability* mechanism that can be used to precisely configure which memory addresses, function interactions, and platform resources each thread can access. Similar to the concept of separation kernels long used to provide foundations for security [14], the precise formally-proven partitioning and information flow control that seL4 provides make it easier to include components of mixed criticality, to "sandbox" untrusted components, and to update portions of the system while ensuring that other parts can never be impacted by the changes. Building on the seL4 foundation, HAMR on CASE supports model-driven development for refactoring systems to achieve greater cyber-resiliency including automated wrapping of legacy components in virtual machines (VM) and automated insertion of high-assurance components such as message filters, network guards, and security health monitors.

In CASE, AADL is used to model system architecture which is automatically analyzed for cyber-resiliency properties and to capture architecture transformations that insert high-assurance components and VMs. HAMR translates an AADL architecture to produce configurations of the seL4 micro-kernel (especially the capability protection specifications), Specifically, HAMR translates AADL system architectures to seL4 architecture description language, called CAmkES [11], along with some C glue code to interface HAMR C AADL RTS with CAmkES. CAmkES is designed to make it easier to configure seL4 capabilities to align with component structures, and is rather agnostic to the particular computational model. The CAmkES framework has its own mechanism to generate low-level C kernel code as well as the seL4 "capDL" (Capability Distribution Language) file. These artifacts together with the kernel itself and CAmkES component code are used create a binary image which can be loaded onto an appropriate processor.

Leveraging the CAmkES code generation, HAMR generates CAmkES declarations to align with AADL, which, via the CAmkES code generation, configures seL4 so that AADL intercomponent information flow pathways are enforced by the microkernel. HAMR also generates additional code that adapts CAmkES threading and communication APIs to align with the AADL RTS and computational model. This includes generating infrastructure code that uses, e.g., seL4 protected shared memory to realize event buffering and dispatch logic of AADL RTS as implemented in the HAMR reference implementation. This is crucial for enabling AADL-level analysis and verification to be sound with respect to generated seL4 deployments. Leveraging the HAMR translation factored through the Slang reference model, the developer-facing C communication APIs and thread skeletons are identical to those generated for Linux (as described in the previous section). We are working with Data61 engineers to implement dedicated CAmkES connectors that realize the AADL semantics – thus, eliminating the need for a layer of adapter code used in the current code generation process.

HAMR also generates virtual machine configurations for CAmkES components that are used to host Linux VMs, e.g., for sandboxing legacy or less trusted code. This ensures that communication across VM boundaries also aligns with AADL communication semantics.

5 Applications

We have applied HAMR to several examples on multiple industrial research projects sponsored by US Department of Homeland Security (DHS), US Air Force Research Labs, US Army, and US Defense Advanced Research Project Agency (DARPA). Below are two examples chosen for their scale, complexity, their coverage of different platforms, and the use of different programming language for code application logic.

5.1 PCA Pump – JVM Platform

The DHS-sponsored ISOSCELES project provides an open-source reference architecture for building medical devices [6]. The project supports device manufacturers and regulatory science by providing freely available resources that incorporate best-practices in MBE as well as architectures designed for safety and security. To validate the ISOSCELES reference architecture, the project utilized example medical device development artifacts from the Open PCA Pump Project (see [9] for an overview of the project and [19] for the project web-site that provides the open-source artifacts). PCA pumps are bedside devices used to infuse opioids into the IV line of a patient. Though the use of PCA pumps is wide-spread, they suffer from safety and security problems. In collaboration with engineers from the US Food and Drug Administration, the Open PCA Pump project developed a collection of realistic open-source development artifacts including an AADL model-based-development implementation of a pump.

The Open PCA Pump AADL model is one of the most complex publicly available AADL models. Just considering thread components and their interactions (excluding other component types not related to code generation), there are 12 thread components, 186 thread component ports, and 101 connections between thread ports. We used HAMR to develop a Slang-based JVM implementation of the pump software along with Slang, Scala, and Java-based simulations of several hardware elements of the pump, including the pump mechanism, fluid flow rate sensors, and operator interface. The resulting system has 14223 non-comment/space source lines of Slang/Scala code (NCSLOC) in the auto-generated infrastructure code and 1220 NCSLOC for the application logic.

5.2 UAV System – seL4 Platform

This example from the DARPA CASE program demonstrates HAMR's ability to support mission systems on a complex high-assurance partitioning platform. The CASE example is intended to demonstrate how CASE technology can harden legacy mission control software for unmanned air vehicles against cyber-attacks. AFRL's open-source UxAS framework, written in C++ using a publish/subscribe communication infrastructure, was used as the existing system to be hardened. A ground station communicates with a surveillance UAV. Before the UAV is launched, map information including operating regions and no-flys zones are loaded into the system. During the course of the mission, the UAV comes into contact with multiple ground stations who receive status information from the UAV and may send updated mission objectives. Mission objectives are processed by a flight planner module to produce collections of waypoints that are fed to the flight computer.

In the first step of the cyber-resilience hardening, the UxAS was broken into pieces to isolate different portions of the system to protect against intrusions and contain the effects of possible Trojan attacks. In this process, the communication stack on board the UAV was migrated into a Linux virtual machine in an seL4 partition. Similarly, the mission planner subsystem (which takes mission commands and map information and computes sets of waypoints for the flight controller) was also migrated to a Linux VM. Both of these are modeled as AADL processes (representing the spatially isolated functionality) bound to an AADL virtual process (representing a virtual machine). AADL properties on these components provide further configuration information. Next, various cyber-resiliency components auto-generated from high-level CASE formal specifications were inserted to filter messages coming from the untrusted legacy components and to monitor (and recover from) sequences of events from the legacy components that suggest that they have been compromised. These components, written in C or CakeML, each run on "bare metal" within their own seL4 partitions. The UxAS Waypoint Manager (which takes a collection of waypoints and feeds individual waypoints to the flight controller as the flight progresses) was considered to be trusted. Its existing C++ implementation was migrated to a bare metal seL4 partition with some hand-written C adapters at the boundaries to align the code with HAMR-generated C port APIs.

During the early phases of this effort, HAMR was first used by the industry team to build a JVM-based prototype of the system where component behaviors were first mocked up in Slang. Once interface design, data types, and other integration issues were solved, HAMR was used to generate a Linux prototype of the system which was refined to include more of the C-based implementations of the system components. Next HAMR was used to generate a fully functional seL4-based deployment (including VMs) for the QEMU simulation environment. Finally, HAMR was used to generate a deployment for seL4 running on an ODroid embedded platform. Thus, even though the initial CASE program goals did not seek to leverage the "multi-platform" nature of HAMR, the ability to quickly build Slang/Scala/Java JVM-based prototypes ended up being quite valuable. Industrial engineers are interested in continuing this approach in future phases of the program. Due to restrictions and proprietary information, we are unable to give precise metrics on the models and code base. The application code size is significantly greater than the other examples.

6 Related Work

The most closely related works to this paper are other AADL code generation frameworks.

Ocarina: Ocarina, led by Hugues [12], is the oldest 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 [16]. 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. It also has a backend for POK, a partitioned operating system compliant with the ARINC653 standard, along with configuration file generation for ARINC653-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's 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 has a focus in integrating code generation for RTOS 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) are prioritizing the use of the machine verified seL4 micro-kernel. HAMR is being used in conjunction with Adventium Labs FASTAR AADL temporal analysis and schedule-generation tools.

RAMSES: The code generation approach of Refinement of AADL Models for Synthesis of Embedded Systems (RAMSES) [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 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 micro-kernel-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.

Trusted Build: HAMR can be seen as a successor to the Trusted Build (TB) concept prototype [7] developed in the DARPA High Assurance Cyber Military Systems (HACMS) Program by Collins Aerospace, University of Minnesota, and Data61. Like HAMR, TB generated component skeletons for seL4 from AADL using the Data61 CAmkES seL4 component modeling language. TB was the first AADL-to-seL4 translation framework. It was used in DARPA HACMS to construct several systems of roughly the same complexity as the UAV system described in Section 5.2.

HAMR provides significant functionality beyond TB. HAMR's port-based inter-component communication strategy now provides true one-way communication from the sender to the receiver on an AADL connection. With TB it was possible to have some back-flow of control and data information, which is undesirable from an information assurance perspective. The TB CAmkES patterns also had unnecessary complexity that require more complex information assurance arguments. In addition, the TB port-based communication structure introduced an extra thread for each connection, dramatically increasing the number of CAmkES components and associated support threads of the generated system, which vastly increases overhead. For example, if one considers deploying the small PCA Pump (Section 5.1) to seL4, the TB approach would generate 101 additional CAmkES components and threads compared to HAMR.

The TB generated structures also did not support standard AADL semantics for ports, so standard model-analysis results did not apply to the implementation. HAMR confirms to the standard, and handles additional AADL features including dispatching strategies (e.g., port urgency, explicit indication of ports that trigger dispatch) and port value freezing. HAMR also supports automated VM building, which reduces both manual labor and 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 set up framework.

7 Conclusion

HAMR is a new open-source multi-platform framework for model-driven development of cyber-physical systems using AADL. The framework has been vetted on a number of government/industry projects in both the medical device and mission control domains. HAMR complements existing AADL code generation tools like Ocarina and RAMSES by supporting additional industry-relevant platforms, and by providing an architecture designed for extensibility. The HAMR theme of supporting industry workflows through a progression of rapid prototyping to deployments on successively realistic platforms is also a new emphasis. HAMR significantly improves on the previous Trusted Build work and compared to other AADL code generation frameworks it provides a distinct area of emphasis: code generation for micro-kernels. Not only does this expand the opportunity to support rigorous CPS development, the experience with additional platforms and code generation architecture are providing inputs to the AADL standards committee for a re-design of the AADL run-time services and code generation annex in the upcoming major version of AADL (the organizations of the authors of this paper have a record of strong and extended participation in the AADL standards committee).

The HAMR approach is *not intrinsically tied to AADL*. Instead, it is linked to the paradigm of real-time tasking in communication in AADL – a paradigm based to real-time tasking models presented in classic textbooks on analyzeable real-time systems [5] and on communication approaches used in avionics buses like ARINC653. Thus, it is possible to replace the AADL front-end with other modeling frameworks that can be aligned with or instantiated to the computational paradigm of AADL. Our current Army SBIR Phase II research project is prototyping a SysML front-end for HAMR, based on the idea of defining a AADL-aligned profile for SysML. This can ease adoption of HAMR for companies that have significant investments in SysML tooling and find it challenging to integrate a different modeling language (AADL) and associated editors.

On a more foundational front, we are leveraging the layered design of HAMR to support the generation of evidence that generated code conforms to the AADL architecture. Aligning with the information assurance emphasis of some of our industrial and defense-related research projects, we are first tackling providing evidence of preservation of model-level information flow and spacial separation, e.g., as visualized in the Awas AADL information flow visualization tool [17]. We are also investigating framework for establishing stronger behavioral correspondence between lower-level generated code and the HAMR reference implementation abstraction layer in Slang.

Regarding the track theme *Programming: What's Next?*, HAMR emphasizes an approach where a modeling language (AADL) and a programming language (Slang, C, etc.) work hand-in-glove to provide the system implementation. The programming language is not used to code all of the system. Instead the model provides a high-level specification of inter-component communication and threading threading structure. Generative techniques are then used to generate a large amount of code. This is similar to other framework approaches like Spring that include high-level specifications of (a) building blocks (abstractions) from which code is derived for common services (b) integration of functionality specified with conventional source code.

What is different for HAMR is the is use of this type of framework for real-time and embedded systems, and in particular the use of building blocks that can be given a formal semantics. As a consequence, reasoning about the correctness of the system is done by reasoning about application source code together with the semantics of the integration abstractions. Given that the code generation is correct with respect to the semantics of the abstractions, the developer nor the verification tools need to be concerned with the details of the infrastructure code. Rather the can rely on the semantic properties of the abstractions.

Slang is not essential for the approach. One can also take this approach with C, for example. However, the use of Slang eases the verification of the application code. Moreover, since the infrastructure code and code generators are written in Slang, HAMR provides the convenience of a single verification framework to establishes the correctness of code generation for abstractions (done once) and the application code (done for each system).

From a bottom-up perspective, HAMR provides a significant contribution by layering application-oriented abstractions on top of the formally verified seL4 micro-kernel – thus providing the foundation for eventually scaling up the formally correctness proofs from the kernel to applications/systems programmed on top of the kernel.

In general, we believe that use of model/code frameworks based on formally-verified domain-specific abstractions with integrated semantics is important direction for engineering critical systems.

Acknowledgements: The authors wish to thank other DARPA CASE team members from Collins Aerospace, Adventium Labs, and Data61 for their work on applications of HAMR and their inputs on HAMR design.

References

1. SAE AS5506/2. AADL annex volume 2
2. Aerospace Vehicle Systems Institute: Motivation for advancing the system architecture virtual integration program. URL (2020), `https://savi.avsi.aero/about-savi/savi-motivation/`
3. AVSI: System Architecture Virtual Integration (SAVI) Initiative (2012)
4. Borde, E., Rahmoun, S., Cadoret, F., Pautet, L., Singhoff, F., Dissaux, P.: Architecture models refinement for fine grain timing analysis of embedded systems. In: 2014 25nd IEEE International Symposium on Rapid System Prototyping. pp. 44–50 (2014)
5. Burns, A., Wellings, A.: Analysable Real-Time Systems: Programmed in Ada. CreateSpace (2016)
6. Carpenter, T., Hatcliff, J., Vasserman, E.Y.: A reference separation architecture for mixed-criticality medical and iot devices. In: Proceedings of the ACM Workshop on the Internet of Safe Things (SafeThings). ACM (November 2017)
7. Cofer, D., Gacek, A., Backes, J., Whalen, M.W., Pike, L., Foltzer, A., Podhradsky, M., Klein, G., Kuz, I., Andronick, J., Heiser, G., Stuart, D.: A formal approach to constructing secure air vehicle software. Computer.org **51**, 14–23 (2018). https://doi.org/10.1109/MC.2018.2876051
8. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2013)
9. Hatcliff, J., Larson, B., Carpenter, T., Jones, P., Zhang, Y., Jorgens, J.: The Open PCA Pump Project: An exemplar open source medical device as a community resource. SIGBED Rev. p. 813 (Aug 2019)
10. International, S.: SAE AS5506 Rev. C Architecture Analysis and Design Language (AADL). SAE International (2017)
11. Kuz, I., Liu, Y., Gorton, I., Heiser, G.: CAmkES: A component model for secure microkernel-based embedded systems. Journal of Systems and Software **80**(5) (2007)
12. Lasnier, G., Zalila, B., Pautet, L., Hugues, J.: Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications. In: Kordon, F., Kermarrec, Y. (eds.) Reliable Software Technologies – Ada-Europe 2009. pp. 237–250. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
13. NICTA, Dynamics, G.: sel4 microkernel (2015), `sel4.systems`
14. Rushby, J.: The design and verification of secure systems. In: 8th ACM Symposium on Operating Systems Principles. vol. 15(5), pp. 12–21 (1981)
15. West, A.: Nasa study on flight software complexity. URL (March 2009), `https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf`
16. Zalila, B., Pautet, L., Hugues, J.: Towards automatic middleware generation. In: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008). pp. 221–228 (2008)
17. Sireum Awas website. `https://awas.sireum.org`
18. DARPA CASE Vagrant. `https://github.com/loonwerks/CASE/tree/master/TA5/case-env`
19. Open PCA Pump Project website. `http://openpcapump.santoslab.org` (2018)