Towards an AADL-Based Definition of App Architecture for Medical Application Platforms⋆

Sam Procter, John Hatcliff, and Robby

Kansas State University,
Manhattan, Kansas, USA
{samprocter,hatcliff,robby}@ksu.edu

Abstract. There is a growing trend of developing software applications that integrate and coordinate the actions of medical devices. Unfortunately, these applications are being built in an ad-hoc manner without proper regard for established distributed systems engineering techniques. We present a tool prototype based on the OSATE2 distribution of the Eclipse IDE that targets the development of Medical Application Platform (MAP) apps. Our toolset provides an editing environment and translator for app architectures, i.e., their components and connections. The toolset generates interface definitions and glue code for the underlying MAP middleware, and it supports development of the business logic which the developer must write to complete the application within the same Eclipse-based environment. We also present a clinical scenario as a motivating example, trace its development through the toolset, and evaluate our work based on the experience.

Keywords: Integrated medical systems, medical application platforms, software architecture, AADL

1 Introduction

Medical devices, which have traditionally been built and certified in a standalone fashion, are beginning to be integrated with one another in various ways: information from one device is forwarded to another for display, information from multiple devices is combined to get a more accurate representation of a patient's health, or a device may even monitor or interrupt routine treatments (e.g., administration of an analgesic) via "closed-loop" execution. The architecture of these systems has begun to attract the attention of regulatory and certification agencies; the FDA now recognizes ASTM's F2761 standard [4], which lays out a functional architecture for an integrated clinical environment (ICE). Implementation of these systems is increasingly realized on some sort of real-time middleware, leading to important realizations about commonalities and differences between them and other, more traditional, distributed systems.

Consider the following scenario which has been identified and studied in the literature that we will use as a motivating example throughout this work [15]:

> After an uneventful hysterectomy, a patient was given a patient-controlled analgesia (PCA) pump to deliver morphine. However, too much of the drug was administered even after her vital signs had become depressed and she ultimately died [5]. The pump could have been disabled and the patient's life saved had a simple monitoring algorithm had access to her physiological monitors and been capable of disabling her PCA pump.

The term medical application platform (MAP) was introduced in [11], where it was defined as a "safety- and security-critical real-time computing platform for: (a) integrating heterogeneous devices, medical IT systems, and information displays via a communication infrastructure, and (b) hosting application programs (i.e. apps) that provide medical utility via the ability to both acquire information from and update/control integrated devices, IT systems, and displays." Similar to [11], we focus on MAPs in a clinical context, though envision their potential use in "portable, home-based, mobile or distributed [systems]."

Although work has been done on application requirements [19] and the architecture of MAPs [11,17], little attention has been paid to the architectures of medical applications (apps) that run on a MAP. These MAP apps are essentially new medical devices that are: (a) specified by an application developer, (b) instantiated at the point of care, and (c) coordinated by the MAP itself. Most of the previously built applications on these prototype platforms were designed in an ad-hoc manner with the goal of demonstrating certain functionality concepts.

What is needed is to move from this ad-hoc approach to something that can enable systematic engineering and reuse. Such an approach would enable true component-based development, which could utilize network-aware devices as services on top of a real-time, publish-subscribe middleware. These components (and their supporting artifacts) could be composed by an application at runtime to define the medical system's behavior, though this would require careful reasoning about the architecture of the application.

While this sort of careful reasoning about MAP apps (also referred to as *virtual medical devices*) has not been performed before, the architecture of other bus-based, safety-critical systems has been given a great deal of attention. The *Architecture Analysis & Design Language* (AADL) was released in November 2004 by the Society of Automotive Engineers (SAE) as aerospace standard AS5506 [8]. Described as "a modeling language that supports early and repeated analyses of a system's architecture…through an extendable notation, a tool framework, and precisely defined semantics," it enables developers to model the architecture of the hardware and software aspects of a system as well as their integration.[1] That is, not only can processors and processes be modeled, but so can the mapping from the latter to the former. It has found some success in systems development in both the aerospace (e.g., the "integrate then build" approach undertaken in the SAVI effort [10]) and automotive fields, and it has even been applied to the development of traditional medical devices [14,21].

---

[1]   Moreover, there are language annexes which enable the specification of a system's behavior in AADL.

Since MAP app development is in need of more engineering rigor and AADL was developed to provide architectural modeling and analysis for safety-critical systems, it seems natural to evaluate their combined use. Additionally, AADL has an established community and has been hardened by nearly a decade of use (see Section 3.2). However, it is not immediately clear whether a technology aimed at the integration of hardware and software in the automotive and aerospace industries will be applicable to the domain of MAP apps. For example, since MAPs do not expose the raw hardware of their platform — rather programming abstractions above it — it is unclear how well certain AADL features will work when only these software abstractions are being modeled, or if they will be necessary at all.

What is needed, then, is: (a) a subset of AADL that is useful when describing the architecture of MAP apps, and (b) a supporting set of tools to facilitate app development. We describe an approach and prototype toolset using a publicly available, open source MAP called the Medical Device Coordination Framework (MDCF [2]); while being specific to MDCF, we believe our work generalizes to other rigorously engineered MAPs.

Specifically, the main contributions of this paper are:

1. A proposed subset of the full AADL (selected components and port-based communication) that is useful for describing a MAP app's architecture.
2. A proposal for a set of properties necessary for describing the real-time (RT) and quality-of-service (QoS) properties of MAP apps. This set includes some of AADL's built-in properties, and it also utilizes AADL's property description mechanism to specify new properties.
3. An implementation of a translator that takes as input the relevant properties and app component structure (as identified in 1 and 2) and produces as output an application context for the MDCF. Specifically, the translator produces code that automatically: (a) configures the underlying publish / subscribe middleware of the MDCF, (b) configures the components to work together as described in the architectural model, and (c) enforces the RT and QoS parameters via properties described in (2).
4. A runnable app which demonstrates the expected translator output and implements the previously discussed clinical scenario. The architecture of the app is specified in our proposed subset of AADL and the output is runnable on the MDCF.

The remainder of this paper is organized as follows. Section 2 outlines our overall app development vision. Section 3 provides a background on technologies and problems relevant to this work, chiefly MAPs and AADL. Section 4 gives a walkthrough of the supported language subset and relevant properties. Finally, Section 5 concludes and describes future work.

2   Vision

At the center of our long-term MAP app development vision is an App Development Environment (ADE) that is built on an industrial grade Integrated Development Environment (IDE) which supports model-driven development of apps. The IDE should provide access to traditional software development tools includ-

ing editors, compilers, debuggers, and testing infrastructure. The ADE should also have a pluggable architecture so that a variety of components for supporting app analysis, verification, and artifact construction can be easily added to the environment. Additionally, the envisioned ADE should enable compilation and packaging of an app so that it can be directly deployed and executed on a MAP such as the MDCF.

In addition to supporting traditional development, an important aspect of our vision is that the ADE should support preparation of a variety of artifacts required in third-party certification and regulatory approval of the app. For example, the ADE should support the construction of requirements documents with capabilities enabling requirements to be traced to both individual features and formal specifications in the app model and implementation. The ADE should support preparation of hazard and risk analysis artifacts (which should also be traceable to models/code). We envision a variety of forms of verification such as (a) app to device interface compatibility checking, (b) component implementation to component contract conformance verification, (c) model checking of real-time aspects of concurrent interactions, (d) error and hazard modeling using the EMV2 annex to AADL [18], and (e) proof environments that support full-functional correctness proofs for an app. The ADE should also support construction of rigorous styles of argumentation, such as assurance cases (again, with traceability links to other artifacts).

The ADE should also support preparation of third-party certification and regulatory submission documents (e.g., the FDA 510(k)), as well as the packaging of artifacts into digitally-signed archives that would be shipped to relevant entities. These organizations would be able to use the same framework to browse the submitted artifacts and re-run tests / verification tools. The work presented in this paper represents the first step towards achieving this vision.

3   Background

Our work on this topic was guided by our experiences with the state of medical application platforms (primarily the MDCF), interest in AADL, previous work with safety-critical software engineering and the associated regulatory oversight, and requirements engineering for MAP apps [19].

3.1   Medical Application Platforms

The concept of MAPs predates the term; one notable publication is the ASTM standard F2761, which introduces one possible MAP architecture, referred to as the *Integrated Clinical Environment* (ICE) [4]. A logical view of an app running on the ICE architecture is given in Figure 1. The ICE manager, which consists of a supervisor and network controller, provides a configurable execution context upon which apps can run and is supported by a real-time middleware that can guarantee various timing and QoS properties. Devices connected to an ICE system can be used (or controlled) by the apps running on the manager, and various services (e.g., logging and display functionality) are provided transparently to the app. There are a number of implementations of the ICE architecture including the MDCF [16] as well as commercial offerings like Dräger's BICEPS [23].

MAP Applications: MAP applications, as distributed systems, are built using the traditional "components and connections" style of systems engineering. Any development environment for apps, then, should have a number of *core features* that are important to component-based development:

- *Support for well-defined interfaces:* The components of distributed systems should be self-contained, with the exception of the inputs and outputs they expose over well-defined interfaces. This enables a library of commonly used components to be made available to the developer.

- *Support for common patterns of communication:* Not only are the components of such a system often reusable, but so are the styles of communication between the components. Adhering to common patterns will also result in a more straightforward software verification process.

- *Support for real-time and quality-of-service configuration:* In a real-time, safety-critical distributed system, correctness requires not only the right information, but also getting it at the right time. Safety arguments can be difficult to make without the ability to set expected timings in an app's configuration (e.g., a task's worst-case execution time) and have a guarantee of the enforcement of those timings from the underlying middleware.

The translator and example artifact portions of this work target the MDCF because it supports a rigorous notion of component specification. Since the MIDdleware Assurance Substrate (MIDAS) [3] is one of the middleware frameworks supported by the MDCF, our translator supports setting a range of timing properties attached to both connections (e.g., a port's latency) and components (e.g., a task's worst-case execution time). As previously noted, though, the work described here is not deeply tied to the MDCF but could be targeted to any MAP implementation that supports similarly rigorous notions of component definition, configuration, and communication.

We believe the *core concepts* common to definitions of app architectures are:

- *Layout:* A high-level schema that defines how various components connect to one another and collectively form the app.

- *Medical Device:* Individual medical devices which will either be controlled by software components, or produce physiological information that will be consumed by software components.

- *Software Component:* Software pieces that typically (though not exclusively) consume physiological information and produce clinically meaningful output: information for a display, smart alarms, or commands to medical devices.

- *Channel:* Routes through a network over which components (i.e., both medical devices and software) can communicate.

Taken together, the core features and concepts enable reusability by ensuring that components communicate over interfaces in common, pattern-based ways with strict timing constraints. A component is capable of interoperating in any other system where it "fits." That is, its interface exposes the required ports, utilizes the same communication patterns, and has compatible timing requirements.

AADL Construct	MAP Concept	Mapping Explanation
Components		
System	Layout	This equivalence is not a large stretch, as systems "[represent] a composite that can include…components" [7]
Device	Device	This is essentially a direct equivalence.
Process	Software Component	AADL processes "[represent] a protected address space that [prevent] other components from accessing anything inside." [7] Tasks are local to the class they're created in, allowing them to share state, but stopping them from directly manipulating tasks or state outside their own class.
Thread	Task	Tasks (which extend `java.lang.runnable`) represent some unit of work to be done either periodically or upon arrival of a message on a designated port.
Connections		
System port connection	Channel	Channels enable communication between components, using a messaging service.
Process port connection	Task Trigger	The only currently supported process-level port types are `data` and `event data`. Message arrival on an `event data` port triggers an associated task, while `data` ports simply update a predictably-named field.
Process impl.-level port connection	Task-Port Communication	A port connection from a process to a thread translates to a task's use of data arriving via that port.

Table 1. AADL syntax elements and their MAP app mappings

3.2   Architecture Analysis & Design Language

SAE's AADL is a standardized, model-based engineering language that was developed for safety-critical system engineering. Therefore, it has a number of features that make it particularly well suited to our needs:

- *Hierarchical refinement:* AADL supports the notion of first defining an element and then further refining it into a decomposition of several subcomponents. This will not only keep the modelling elements more clean and readable, but will also allow app creators to work in a top-down style of development. They will be able to first think about what components make up the system and how those components would be linked together, define those components, and finally reason about how those individual components would themselves be comprised.
- *Distinction between types and implementations:* AADL allows a developer to first define a component and then give it one or more implementations, similar to the object-oriented programming practice of first defining an interface and then creating one or more implementations of that interface. This keeps app models cleaner and enables code re-use.
- *Extensible property system:* AADL allows developers to create properties, specify their type, and restrict which constructs they can be attached to. We have used this feature to, for example, associate various physiological parameters with their IEEE11073 nomenclature "tag" [12].
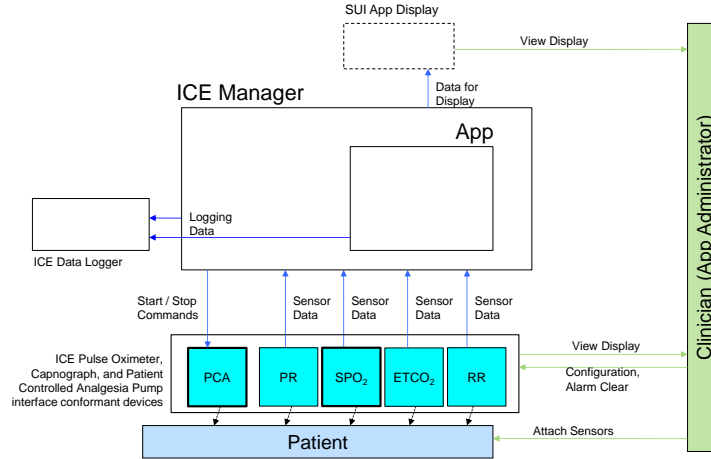
Fig. 1. The PCA Interlock App's information flows. The devices used in the app excerpt discussed in this section have bolded borders.

- *Textual and graphical representations:* AADL has a defined textual and graphical syntax, and there is tool support for converting between the two representations.
- *Strong tool support:* AADL is supported by a wide range of both open source (e.g., OSATE2 [22]) and commercial (e.g., STOOD [6]) tools. We have used OSATE2 as the basis for our toolset, and have found a number of its features quite useful (e.g., element name auto-completion, type-checking, etc.).

In general, AADL models are composed of components and their connections. AADL includes a number of software modeling entities (e.g., thread, process, data, etc.), hardware entities (e.g., processor, memory, device, etc.), and composite entities (e.g., system and abstract). AADL also includes connections between components of various types such as ports, data accesses, bus accesses, etc. Of these, our translator uses only a small subset: the system, device, process and thread entities, as well as port communication. Elements outside of this subset are either not needed for app construction (e.g., the flows entity) or may be added later (e.g., the subprogram entity). The mapping from these constructs to the MAP constructs identified in Section 3.1 is listed in Table 1; a full explanation of how our subset is used to define an app is given in the next section.

4   Language Walkthrough

In this section, we describe the process of creating a MAP app with our prototype toolset using the motivating example of the PCA Interlock app initially described in Section 1. This app consumes various physiological parameters to determine the health of the patient and disables the flow of an analgesic when there are in-
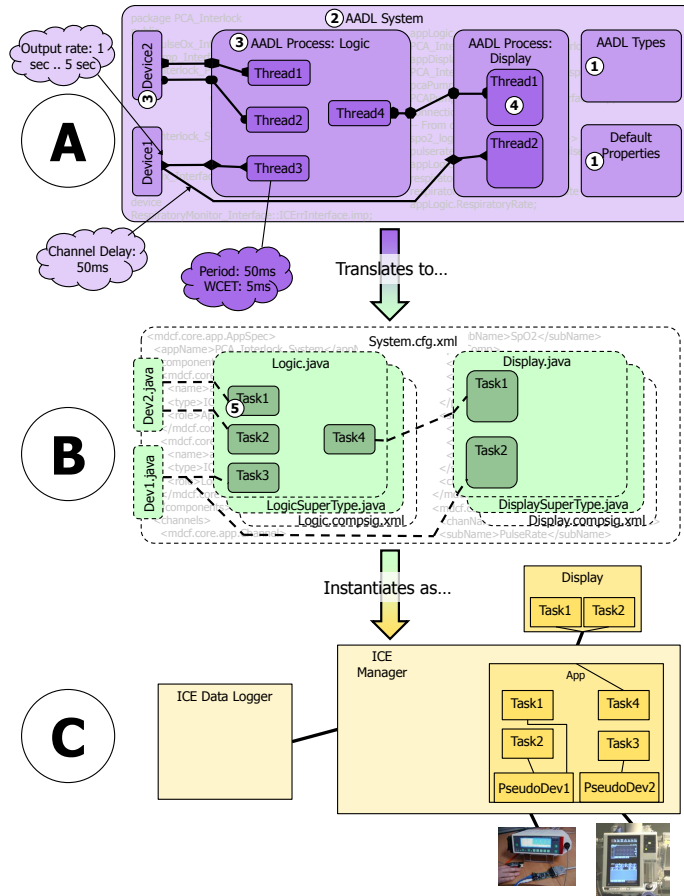
Fig. 2. (A) The AADL platform artifacts used by the code generation process, (B) the generated app configuration and executable files, and (C) the fully configured and executing platform.

dications of respiratory failure [20]. A high-level, ICE-configuration/logical view of the app is shown in Figure 1. The diagram shows that in addition to the PCA pump, there are four sensors: (i) a blood-oxygen saturation (SpO_2) sensor, (ii) a pulse rate sensor, (iii) a respiratory rate sensor, and (iv) an end-tidal carbon dioxide ($ETCO_2$) sensor. In this application, the sensors may be on the same device: SpO_2 and pulse rate information are often produced by a pulse oximeter (e.g., the Ivy 450C [13]), and respiratory rate and $ETCO_2$ information can come from, e.g., a capnography machine (e.g., the Capnostream 20 [1]).

The PCA pump consumes information from the app (e.g., enable and disable commands) while the others produce information in the form of sensor data that is used by the app's logic. An important part of the app (and the underlying MAP/MDCF) is that it will use suitable physiological parameters regardless of their source; that is, instead of building the app to work with a specific device

```
1  package PCA_Interlock_Types
2  public
3  with Data_Model, IEEE11073_Nomenclature;
4    data SpO2
5    properties
6      Data_Model::Data_Representation => Integer;
7      IEEE11073_Nomenclature::OID => IEEE11073_Nomenclature::MDC_PULS_OXIM_SAT_O2;
8      Data_Model::Integer_Range => 0 .. 100;
9    end SpO2;
10 end PCA_Interlock_Types;
```

(a) The SpO$_2$ datatype used in the app excerpt

```
1  property set PCA_Interlock_Properties is
2    with PCA_Interlock;
3    Default_Thread_Time : constant Time => 50 ms;
4    Default_Output_Rate : Time_Range => 100 ms .. 300 ms applies to (port);
5    Default_Thread_Dispatch : Supported_Dispatch_Protocols => Sporadic applies to (
         thread);
6    Default_Thread_Period : Time => PCA_Interlock_Properties::Default_Thread_Time
         applies to (thread);
7    Default_Thread_Deadline : Time => PCA_Interlock_Properties::Default_Thread_Time
         applies to (thread);
8    Default_Thread_WCET : Time => 5 ms applies to (thread);
9    Default_Channel_Delay : Time => 100 ms applies to ({PCA_Interlock} ** port
         connection);
10 end PCA_Interlock_Properties;
```

(b) The default properties used in the app excerpt

Fig. 3. Data types and default properties used in the app excerpt

or set of devices, it is built to work with a generic source of the required physiological parameters. We hope to present the process for deriving/documenting the requirements (e.g., [19]) that were used to create this diagram/drive the app development process in future work.

Figure 2 gives an overview of the app architecture development, code generation, and app instantiation process. Part (A) of the figure shows the various AADL artifacts that compose the app; note that they are labeled with the number of the subsection they are discussed in. Part (B) shows the execution and configuration artifacts that result from code generation. It also highlights the large number of components (signified by dashed lines) that are fully automatically generated. Part (C) shows the app's instantiation on a running MAP, which was first sketched in Figure 1, with both the computation hosting and communication aspects of the app having been realized in the ICE architecture.

This section presents excerpts of AADL models that specify the application architecture which, when used with our translator, results in application code runnable on the MDCF. Due to space constraints, we only show one physiological parameter: SpO$_2$, though the full app would contain all four parameters (i.e., SpO$_2$, pulse rate, respiratory rate, and ETCO$_2$). In the next section, we discuss AADL types and default properties, followed by a top-down walkthrough of the hierarchy of components used by our toolset.

4.1 Preliminary tasks: Types and Default Properties

Before we can describe a MAP app's architecture, we must briefly examine AADL's type and property definition mechanisms (marked by a (1) in Figure 2) and how they are used to specify various parameters in our app.

Default Name Override Name	Type	Example	Explanation
Thread Properties			
Default_Thread_Period Timing_Properties::Period	Time	50 ms	Periodic tasks will be dispatched to run once per period.
Default_Thread_Deadline Timing_Properties:: Deadline	Time	50 ms	A task will be scheduled such that it has time to complete before its deadline.
Default_Thread_WCET Timing_Properties:: Compute_Execution_Time	Time	50 ms	A task's worst case execution time is the most time it will take to complete after dispatch.
Default_Thread_Dispatch Thread_Properties:: Dispatch_Protocol	Sporadic or Periodic	Periodic	Periodic tasks are dispatched once per period, sporadic upon message arrival.
Port Properties			
Default_Output_Rate MAP_Properties:: Output_Rate	Time Range	100 ms .. 300 ms	Ports must specify the most and least frequently that they will broadcast a message.
Port Connection Properties			
Default_Channel_Delay MAP_Properties:: Channel_Delay	Time	100 ms	Specifies the maximum time that the message can spend on the network.
Process Properties			
N/A MAP_Properties:: Component_Type	Logic or Display	Display	Processes are either for logic or display components.

Table 2. AADL property names, types, examples, and explanations

Data Types: The data type for the SpO_2 parameter is shown in Figure 3a. AADL's property description mechanism is easily extensible, allowing us to specify customer-specific metadata. In this example, we have leveraged this capability to attach an IEEE11073 nomenclature "tag" with our SpO_2 parameter [12]. Note that these datatypes could either be generated from or mapped down to a more standard interface definition language (e.g., CORBA IDL [24]).

Default property values: While it is useful to be able to attach properties to individual AADL constructs (e.g., ports, connections, threads, etc.), sometimes a large number of constructs take the same values for certain properties. In this case, it is useful to set app-wide defaults, as shown in Figure 3b. These properties apply to every applicable element, unless overridden. A full listing of default and override property names and types is shown in Table 2.

4.2   The AADL System

The top level of the app architecture is described by an AADL `system`, marked by a (2) in Figure 2, which is shown textually in Figure 4. Systems have no external features (lines 4-5), though the `system implementation` lists their internals (lines 7-20). An AADL system implementation consists of a declaration of sub-components (e.g., devices and processes), the connections between

```
1  package PCA_Interlock
2  public
3  with SpO2Req_Interface, PCAPump_Interface, PCA_Interlock_Logic,
       PCA_Interlock_Display;
4    system PCA_Interlock_System
5    end PCA_Interlock_System;
6
7    system implementation PCA_Interlock_System.imp
8    subcomponents
9      spo2Device : device SpO2Req_Interface::SpO2Interface.imp;
10     appLogic : process PCA_Interlock_Logic::ICEpcaInterlockProcess.imp;
11     appDisplay : process PCA_Interlock_Display::ICEpcaDisplayProcess.imp;
12     pcaPump : device PCAPump_Interface::ICEpcaInterface.imp;
13   connections
14     -- From components to logic
15     spo2_logic : port spo2Device.SpO2 -> appLogic.SpO2;
16     DisablePump_logic : port appLogic.DisablePump -> pcaPump.DisablePump
17     {MAP_Properties::Channel_Delay => 50 ms;};
18     -- From components to display
19     spo2_disp : port spo2Device.SpO2 -> appDisplay.SpO2;
20     DisablePump_disp : port appLogic.DisablePump -> appDisplay.DisablePump;
21   flows
22     spo2_flow : end to end flow spo2Device.spo2_flow -> spo2_logic -> appLogic.
         spo2_flow -> DisablePump_logic -> pcaPump.spo2_flow;
23   end PCA_Interlock_System.imp;
24 end PCA_Interlock;
```

Fig. 4. The top-level app excerpt architecture via the AADL system component

them, and optionally a description of the paths (flows) data take through the system. Note that we show only excerpts; other interactions include alarm/alert communication and parameter setting. Flows (line 22 of Figure 4, line 10 of Figure 5, and line 8 of Figure 6) allow a developer to trace the path that data take through an entire system and then compute various timings about them such as their overall latency. Previous work on AADL (e.g. [9]) has identified a variety of different analyses that leverage flow specifications. Part of our work is identifying to what extent existing flow analyses can apply to MAPs, that is, we are investigating how they can be used to reason about local and end-to-end communication latencies, secure information flow properties, and coupling/dependencies between components of different criticality levels.

4.3   The AADL Process and Device

Now that the software and hardware elements — the AADL processes and devices marked by a (3) in Figure 2 — have been referenced by the AADL system implementation, a developer must specify their type and implementations.

AADL Processes: A process defines the boundaries of a software component, and is itself potentially composed of a number of threads (Section 4.4). The type of a process in AADL is a listing of that which is visible to components external to the process, i.e., the ports that other components can use to communicate with this this component (see lines 6-8 of Figure 5). The process implementation is, like the system implementation that was discussed in Section 4.2, a listing of subcomponents and connections. The only valid subcomponents (in our subset of AADL) of process implementations are threads. Similarly, all connections are directional links between a thread and one of the

```
1  package PCA_Interlock_Logic
2  public
3  with PCA_Interlock_Types, PCA_Interlock_Properties, MAP_Properties;
4    process ICEpcaInterlockProcess
5    features
6      SpO2 : in event data port PCA_Interlock_Types::SpO2;
7      DisablePump : out event data port PCA_Interlock_Types::Notification
8      {MAP_Properties::Output_Rate => 1 sec .. 5 sec;};
9    flows
10     spo2_flow: flow path SpO2 -> DisablePump;
11   properties
12     MAP_Properties::Component_Type => logic;
13   end ICEpcaInterlockProcess;
14
15   process implementation ICEpcaInterlockProcess.imp
16   subcomponents
17     UpdateSpO2Thread : thread UpdateSpO2Thread.imp;
18     DisablePumpThread : thread DisablePumpThread.imp;
19   connections
20     incoming_spo2 : port SpO2 -> UpdateSpO2Thread.SpO2;
21     outgoing_disable_pump : port DisablePumpThread.DisablePump -> DisablePump;
22   end ICEpcaInterlockProcess.imp;
23 end PCA_Interlock_Logic;
```

Fig. 5. An AADL process specification used in the app excerpt

```
1  package SpO2Req_Interface
2  public
3  with PCA_Interlock_Types;
4    device SpO2Interface
5    features
6      SpO2 : out event data port PCA_Interlock_Types::SpO2;
7    flows
8      spo2_flow : flow source SpO2;
9    end SpO2Interface;
10
11   device implementation SpO2Interface.imp
12   end SpO2Interface.imp;
13 end SpO2Req_Interface;
```

Fig. 6. An AADL device used in the app excerpt

process's ports. Both logic and display components are modeled as AADL processes, and they are distinguised from one another via the MAP_Properties::Component_Type property (line 12).

AADL Devices: Apps describe the devices they need to connect to using the AADL device construct (see Figure 6). Device components are placeholders for actual devices that will be connected to the app when it is launched. These actual devices will have capabilities (like ports, line 6) that match the declared AADL device component specification. Note that the device implementation is left empty, since the app's device needs can be met by any device that realizes the interface requirements.

4.4   The AADL Thread

AADL threads, marked by a (4) in Figure 2, represent semi-independent units of functionality, and are realized in the MDCF as MIDAS tasks. They can be either: (a) sporadic, which signifies that they are executed when a port that they are "attached" to receives a message, or (b) periodic, where they are executed

```
1  thread UpdateSpO2Thread
2  features
3    SpO2 : in event data port PCA_Interlock_Types::SpO2;
4  end UpdateSpO2Thread;
5
6  thread implementation UpdateSpO2Thread.imp
7  end UpdateSpO2Thread.imp;
8
9  thread DisablePumpThread
10 features
11   DisablePump : out event data port PCA_Interlock_Types::Notification;
12 properties
13   Thread_Properties::Dispatch_Protocol => Periodic;
14   Timing_Properties::Period => 50 ms;
15   Timing_Properties::Deadline => 10 ms;
16   Timing_Properties::Compute_Execution_Time => 5 ms;
17 end DisablePumpThread;
18
19 thread implementation DisablePumpThread.imp
20 end DisablePumpThread.imp;
```

Fig. 7. Two AADL thread interfaces used in the app excerpt

after some period of time. Typically, threads which consume information operate sporadically (so they can act as soon as updated data arrive), and threads which produce information operate periodically. Alternatively, ports which are marked as data instead of event data will not trigger any thread execution, but rather will silently update a predictably-named field. This is useful in apps where there are a large number of physiological parameters — rather than specify behavior to be executed each time a message arrives, the most recent data can simply be used when needed.

Thread implementations are empty because this is the lowest level of abstraction supported; all work below this is considered "behavioral," and thus not implemented in AADL but is instead implemented within code templates auto generated by our translator. Figure 7 shows excerpts of two thread interfaces: the first consumes SpO_2 information as it arrives (lines 1-4), and the second disables the PCA pump as necessary (lines 9-17).

4.5   Concluding tasks: Code generation and Instantiation

At this point, the app's architecture description is complete. The next step is to generate the MAP-compatible, executable code (part B of Figure 2). Our translator will interpret the AADL to create a model of the app, and then render it to a target MAP implementation; currently the only implementation supported is the MDCF. In the MDCF, Java is the language used for execution (see Figure 8) and XML for configuration (see Figure 9).

Execution and Configuration Code: Our app contains several components that acquire current physiological readings, others compute the conditions for shutting off the PCA pump, while others execute the interlock protocol. Figure 8 shows a very simple example of how one acquires the current SpO_2 value and stores it for other components to utilize. Note that there is a great deal of auto-generated code (not shown here due to space constraints) that is hidden from the user in the development process (for e.g., marshalling and un-marshalling messages, task instantiation, and error handling).

```
1   @Override
2   protected void initComponent() {
3       // TODO Fill in custom
            initialization code here
4   }
5
6   @Override
7   protected void
        SpO2ListenerOnMessage(
        MdcfMessage msg, Integer
        SpO2Data) {
8       // TODO Fill in custom listener
            code here
9   }
```

```
1   @Override
2   protected void initComponent() {
3       LatestSpO2 = -1;
4       PreviousSpO2 = -1;
5   }
6
7   @Override
8   protected void
        SpO2ListenerOnMessage(
        MdcfMessage msg, Integer
        SpO2Data) {
9       PreviousSpO2 = LatestSpO2;
10      LatestSpO2 = SpO2Data;
11  }
```

(a) Executable "skeletons" produced by the translator

(b) The same "skeletons" complete with business logic

Fig. 8. Executable code, before and after business logic implementation

Figure 9a shows an excerpt of an app configuration XML file; at app launch the runtime system interprets this file and instantiates the software components. An excerpt of a software component's description is shown in Figure 9b.

When the app is launched, the executable artifacts will combine to define the behavior of the app, and the configuration schematics will describe how the various components communicate. Part (C) of Figure 2 shows how primary elements of the app excerpt would look on an ICE implementation at runtime.

5   Conclusion

As outlined in Section 1, our goal for this effort was to identify "a subset of AADL that is relevant to describing the architecture of MAP applications" and to evaluate our proposal by attempting to construct a MAP app with our toolset and language. We found that while AADL was originally conceived for the aeronautics domain, it is well-suited to the description of MAP app architectures. That said, it is not a perfect fit — not only were there components whose semantics did not line up perfectly with the target domain (e.g., processes, see Section 4.3) but there are also predeclared properties that were defined differently than we needed. Since these properties cannot be redefined, we had to create our own (e.g., `Channel_Delay`). Additionally, there were port communication patterns that were only approximable with a publish-subscribe middleware (i.e., there is no shared memory access).

5.1   Future Work

As with any new proposal, we anticipate considerable iteration of our language and tooling as they mature.

Language Extensions: We are interested in considering extensions to the language to support the numerous features discussed in Section 2, as well as those that would enhance the rigor of the architectural descriptions consumed by our translator, such as a mechanism to specify intraprocess communication (i.e., communication between tasks in the same Java class).

```
1  <appName>PCA_Interlock_System</appName>
2  <components>
3   <VirtualComponent>
4    <name>appDisplay</name>
5    <type>ICEpcaDisplayProcess</type>
6    <role>AppPanel</role>
7   </VirtualComponent>
8   ...
9  </components>
10 <channels>
11  <Channel>
12   <chanName>$PH$</chanName>
13   <pubName>$PH$</pubName>
14   <subName>SpO2</subName>
15   <pubComp>
16    <name>$PH$</name>
17    <type>$PH$</type>
18    <role>Device</role>
19   </pubComp>
20   <subComp>
21    <name>appLogic</name>
22    <type>ICEpcaInterlockProcess</type>
23    <role>Logic</role>
24   </subComp>
25   <channelDelay>100</channelDelay>
26  </Channel>
27  ...
28 </channels>
```

(a) An excerpt of the app's overall layout configuration

```
1  <AppModuleSignature>
2   <type>ICEpcaProcess</type>
3   <moduleTasks>
4    <TaskSignature>
5     <type>PORT_SPORADIC</type>
6     <trigPort>SpO2In</trigPort>
7     <period>-1</period>
8     <name>UpdateSpO2Thread</name>
9     <deadline>50</deadline>
10    <wcetMs>5</wcetMs>
11   </TaskSignature>
12   ...
13  </moduleTasks>
14  <portSignatures>
15   <entry>
16    <string>SpO2In</string>
17    <PortSignature>
18     <name>SpO2In</name>
19     <direction>SUB</direction>
20     <minPeriod>100</minPeriod>
21     <maxPeriod>300</maxPeriod>
22     <type>Integer</type>
23    </PortSignature>
24   </entry>
25   ...
26  </portSignatures>
27 </AppModuleSignature>
```

(b) An excerpt of the logic module's configuration

Fig. 9. Configuration schemata for the app and its logic component

Work with Collaborators: We also continue to interact with our research partners at the Center for Integration of Medicine and Innovative Technology, Underwriter's Laboratory (including on the proposed UL 2800 standard for safety in medical device interoperability), and the US Food and Drug administration to validate our approach and develop guidelines for safety and regulatory reviews.

References

1. Capnostream 20 Bedside Patient Monitor. http://www.covidien.com/rms/products/capnography/capnostream-20p-bedside-patient-monitor.
2. MDCF website. http://mdcf.santos.cis.ksu.edu.
3. Andrew King and Sanjian Chen and Insup Lee. The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems With OpenFlow. In *17th IEEE Computer Society Symposium*, 2014.
4. ASTM International. ASTM F2761 - Medical Devices and Medical Systems - Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE), 2009.
5. R. A. Caplan, M. F. Vistica, K. L. Posner, and F. W. Cheney. Adverse anesthetic outcomes arising from gas delivery equipment: a closed claims analysis. *Anesthesiology*, 87(4):741–748, 1997.
6. P. Dissaux. Using the aadl for mission critical software development. In *2nd European Congress ERTS, EMBEDDED REAL TIME SOFTWARE Toulouse*, 2004.

7. P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. 2012.
8. P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis & Design Language (AADL): An introduction. Technical report, DTIC Document, 2006.
9. P. H. Feiler and J. Hansson. Flow latency analysis with the architecture analysis and design language (aadl). Technical report, Carnegie Mellon University – Software Engineering Institute, 2008.
10. P. H. Feiler, J. Hansson, D. De Niz, and L. Wrage. System architecture virtual integration: An industrial case study. Technical report, DTIC Document, 2009.
11. J. Hatcliff, A. King, I. Lee, A. MacDonald, A. Fernando, M. Robkin, E. Vasserman, S. Weininger, and J. M. Goldman. Rationale and architecture principles for medical application platforms. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 3–12. IEEE, 2012.
12. ISO/IEEE. Domain information model. In *ISO/IEEE11073-10201 Health informatics - Point-of-care medical device communication*, 2004.
13. Ivy Biomedical Systems Inc. Vital-Guard 450C Patient Monitor with Nellcor SpO$_2$, Aug 2005.
14. B. Kim, L. T. Phan, O. Sokolsky, and L. Lee. Platform-dependent code generation for embedded real-time software. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 2013.
15. A. King, D. Arney, I. Lee, O. Sokolsky, J. Hatcliff, and S. Procter. Prototyping closed loop physiologic control with the medical device coordination framework. In *Proceedings of the 2010 ICSE Workshop on Software Engineering in Health Care*, pages 1–11. ACM, 2010.
16. A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.
17. A. L. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. P. Jetley, P. L. Jones, and S. Weininger. A publish-subscribe architecture and component-based programming model for medical device interoperability. *SIGBED Review*, 6(2):7, 2009.
18. B. Larson, J. Hatcliff, K. Fowler, and J. Delange. Illustrating the aadl error modeling annex (v. 2) using a simple safety-critical medical device. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, pages 65–84. ACM, 2013.
19. B. Larson, J. Hatcliff, S. Procter, and P. Chalin. Requirements specification for apps in medical application platforms. In *Software Engineering in Health Care (SEHC), 2012 4th International Workshop on*, pages 26–32. IEEE, 2012.
20. R. R. Maddox and C. Williams. Clinical experience with capnography monitoring for pca patients. *APSF Newsletter*, 26:3, 2012.
21. A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl. Compositional verification of a medical device system. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*. ACM, 2013.
22. S. OSATE. An extensible open source aadl tool environment. *SEI AADL Team technical Report*, 2004.
23. S. Schlichting and S. Pöhlsen. An architecture for distributed systems of medical devices in high acuity environments. Technical report, Dräger, 2014.
24. J. Siegel. *CORBA 3 fundamentals and programming*, volume 2. John Wiley & Sons Chichester, 2000.