9-29-2013

Platform-Dependent Code Generation for Embedded Real-Time Software

BaekGyu Kim
University of Pennsylvania, baekgyu@cis.upenn.edu

Linh T.X. Phan
University of Pennsylvania, linhphan@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Recommended Citation

BaekGyu Kim, Linh T.X. Phan, Oleg Sokolsky, and Insup Lee, "Platform-Dependent Code Generation for Embedded Real-Time Software", *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2013)* , 8.1-8.10. September 2013. http://dx.doi.org/10.1109/CASES.2013.6662512

International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2013), Montreal, Canada, September 29 - October 4, 2013.

Platform-Dependent Code Generation for Embedded Real-Time Software

Abstract

Code generation for embedded systems is challenging, since the generated code (e.g., C code) is expected to run on a heterogeneous set of target platforms with different characteristics, such as hardware/software architectures and programming interfaces. We propose a code generation framework that provides the flexibility to generate different source code that is executable on each target platform. In our framework, the platform-dependent characteristics of a target platform are explicitly specified by an Architectural Analysis Description Language (AADL) model and a code snippet repository. The AADL model captures hardware/software architectural aspects of the platform, such as periodic/aperiodic threads and their interactions with sensors and actuators. The code snippet repository contains platform-dependent code snippets that are categorized according to the functions required to implement the components of the AADL model. These two elements of the platform capability are then used by the code generation algorithm to generate platform-dependent code for the given platform. We demonstrate the applicability of our framework using a case study of code generation for two infusion pump systems.

Keywords

Code generation, Embedded software, AADL, Model-based development

Disciplines

Computer Engineering | Computer Sciences

Comments

International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2013), Montreal, Canada, September 29 - October 4, 2013.

Platform-Dependent Code Generation for Embedded Real-Time Software

BaekGyu Kim Linh T.X. Phan Oleg Sokolsky Insup Lee
University of Pennsylvania
{baekgyu,linhphan,sokolsky,lee}@cis.upenn.edu

ABSTRACT

Code generation for embedded systems is challenging, since the generated code (e.g., C code) is expected to run on a heterogeneous set of target platforms with different characteristics, such as hardware/software architectures and programming interfaces. We propose a code generation framework that provides the flexibility to generate different source code that is executable on each target platform. In our framework, the platform-dependent characteristics of a target platform are explicitly specified by an Architectural Analysis Description Language (AADL) model and a code snippet repository. The AADL model captures hardware/software architectural aspects of the platform, such as periodic/aperiodic threads and their interactions with sensors and actuators. The code snippet repository contains platform-dependent code snippets that are categorized according to the functions required to implement the components of the AADL model. These two elements of the platform capability are then used by the code generation algorithm to generate platform-dependent code for the given platform. We demonstrate the applicability of our framework using a case study of code generation for two infusion pump systems.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Design, Reliability

Keywords

Code generation, Embedded software, AADL, Model-based development

1. INTRODUCTION

Model-based development has gained attention as an effective method for creating safety-critical embedded software, such as medical devices. In this approach, the software behavior is abstracted using a high-level model, such as UPPAAL [5] or Simulink/Stateflow [10], and the correctness of the software is verified by an associated analysis tool, such as UPPAAL model checker or Simulink Design Verifier. Code generation tools, such as TIMES [4] or Real-Time Workshop, are then used to systematically convert the verified model to source code, such as C code, that can be executed on target platforms. This systematic process ensures that the resulting code preserves the properties that have been verified at the model level.

This is a post-print version of the paper appearing in CASES 2013, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems

There exist several semantic gaps between high-level models and implementation platforms in this methodology, however. A model typically abstracts information as to how model elements are implemented on a particular target platform; although this abstraction makes the model easier to understand and reduces verification complexity, it places additional burden on the code generator. Further, modeling languages offer abstract semantics that is not well-matched to the execution of the model on target platforms. For instance, UPPAAL provides semantics of state-transition systems along with channel synchronizations and shared variables, which can be used to express inputs and outputs of the software system. Stateflow provides semantics of events and conditions for a similar purpose. Implementing such semantics on an embedded platform requires platform-dependent information, such as how different threads are scheduled to process input data from sensors, and how results of computation are delivered to actuators through some communication mechanism provided by the underlying hardware/-software stack (e.g., real-time operating system). However, this information is not captured in the code generation.

The above gaps give rise to an integration challenge: the generated code cannot be easily integrated with diverse hardware/software architectures of embedded platforms without violating the behavioral compatibility of the code running on the different platforms. Currently, this integration challenge is being solved through manual effort that is tedious and error-prone, because it involves low-level platform-specific details. Therefore, it is necessary to capture the required platform-dependent aspects – in a succinct manner and at the right level of abstraction – in the code generation to guarantee the behavioral compatibility of the generated code when executing on different target platforms.

As the discussion above shows, we need to distinguish two kinds of platform-dependent information. The first describes the mechanisms used by the platform to execute the platform-independent computation, such as whether threads are executed periodically or aperiodically, whether sensor data are sampled or buffered, etc. The second describes a particular platform API offered by a platform for access to the desired mechanisms, such as what calls the software needs to make to access buffers, to read timers, etc.

We propose a code generation framework that provides the flexibility to generate different source code that is executable on each target platform. In our framework, different platform-dependent aspects are expressed using different *platform capabilities*. A platform capability captures two aspects of a platform. First, the hardware/software architecture of a target platform is modeled using Architectural Analysis Description Language (AADL) [1]. In this process, an AADL model is used to capture interactions between the hardware (e.g., sensors/actuators) and the software (e.g., periodic/aperiodic threads). Second, the programming interface of a target platform is captured by a code snippet repository. The repository provides code snippets that are categorized according to their

functions to implement AADL components. The code generation algorithm takes different platform capabilities as inputs to generate the source code that is executable on the corresponding platforms. In summary, this paper makes the following contributions:

- We propose a way to specify platform capabilities that capture (1) hardware/software architectural aspects, and (2) programming interfaces of target platforms.
- We propose a code generation algorithm that synthesizes different source code executable on different target platforms from the platform capabilities.
- We demonstrate the flexibility of our code generation framework via a case study that generates source code for two infusion pump systems that have different platform capabilities.

The rest of the paper is organized as follows. We explain the notion of platform-dependent software aspects in Section 2. Section 3 presents our approach for platform-dependent code generation. In Section 4 and 5, we describe how to capture the platform capability of a platform using an AADL model and a code snippet repository, followed by the code generation algorithm in Section 6. We demonstrate our approach through a case study in Section 7. We discuss the related work in Section 8 and conclude in Section 9.

2. PLATFORM-INDEPENDENT VS. PLATFORM-DEPENDENT ASPECTS

We begin by giving an example of the platform-independent model for the software of an infusion pump. We first describe the necessary information to execute a platform-independent model on a target platform. We then explain two platform-dependent aspects that will be considered in our code generation framework.

2.1 Example: Platform-independent model

An infusion pump is a safety-critical embedded system that injects drugs into the patient body in a controller manner for various medical treatment purposes, such as pain-relief or insulin therapy. A typical infusion pump has a syringe-type drug reservoir whose movement is controlled by hardware/software mechanisms, so that a precise amount of drug can be delivered to the patients. In addition, infusion pumps detect several alarming conditions, such as empty reservoir or occlusion, in order to perform safe infusion therapy. An example of infusion pump systems is shown in Figure 4.

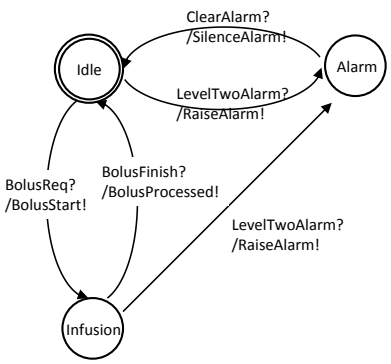

Figure 1: Platform-independent model of an infusion pump.

Figure 1 shows an abstract model of the software behavior of infusion pumps, specified as a state-transition system. The inputs and outputs of the software system are specified on the transitions of the model to express interactions between the system and its environment. A state is modeled using a circle (a double-circle represents an initial state); a transition is modeled using an arrow from one state to another; a question mark (?) indicates an input to the model; and an exclamation mark (!) indicates an output produced by the model. Such a transition system is widely used to model and verify software behavior for many embedded systems (e.g., using UPPAAL and Stateflow).

In this example, a patient may request a certain amount of drug, called a bolus, by pressing a patient-controlled button to initiate the infusion process. This is represented by a transition from the *Idle* state to the *Infusion* state that is associated with *BolusReq?* and *BolusStart!*. When an infusion pump encounters some alarming conditions when infusion is in progress, such as an empty reservoir condition, then an alarm should be raised. This is represented as a transition from the *Infusion* state to the *Alarm* state that is associated with *LevelTwoAlarm?* and *RaiseAlarm!*.

The model shown in Figure 1 is a *platform-independent* model in a sense that it does not explicitly contain information about how it should be executed on target platforms. For example, different target platforms can implement the semantics of *BolusReq?* differently: one target platform may implement a periodic thread that samples the status of electrical signal level of the patient-controlled button, whereas another target platform may implement an aperiodic thread that is invoked upon interrupt-trigger when it detects the change of the signal level of the button; these two platforms equally have a capability to handle the input, but in a different way.

2.2 Platform-dependent software aspects

Our goal is to propose a code generation framework that generates different source code for running on different target platforms. In this framework, the following aspects of a platform are specified separately:

1. the platform-dependent mechanism to execute the platform independent computation;

2. the platform-dependent API provided by a given platform for implementing the platform-dependent mechanism.

Different target platforms may have different mechanisms to perform the same platform-independent computation. One platform may execute the platform-independent computation as a periodic thread, which periodically reads input, computes transitions, and writes output. However, another platform may execute the same computation as an aperiodic thread, which performs the computation only when input arrives. For example, the platform-independent model in Figure 1 may periodically read inputs for *BolusReq?*, *BolusFinish?*, *LevelTwoAlarm?*, *ClearAlarm?*, and then take the corresponding outgoing transitions and write outputs. However, the model may also read those inputs aperiodically; that is, the platform-independent computation is performed only if any of those events occurs. In our framework, such differences in executing the platform-independent computation are captured using AADL models, from which different source code running for the target platforms can be generated.

In addition, different target platforms may have different software stacks (e.g., real-time operating system) that provide different APIs to access to the platform-dependent mechanisms. As an example, Listing 1 shows the code snippet that implements a periodic thread for empty reservoir detection running on FreeRTOS [3]. The code snippet includes several API calls that are provided by FreeRTOS. *TaskCreate* API (Line 8) is called to register necessary information to the OS kernel such as thread's priorities and callback functions. *vTaskDelayUntil* API (Line 17) is called to block the callback function *cbEmptyRsv* until the next invocation period (in this example, the period is 500 ms).

Listing 1: Code snippet of a periodic task in FreeRTOS

```
1  //Declaration part
2  const portTickType periodEmptyRsv=500;
3  const portBASE_TYPE prioirtyEmptyRsv=2;
4  const portBASE_TYPE stacksizeEmptyRsv=500;
5
6  //Initialization part
7  void init_EmptyRsv( void ){
8   TaskCreate(cbEmptyRsv,''EmptyRsv'',stacksizeEmptyRsv,
        NULL,prioirtyEmptyRsv,NULL);
9  }
10
11 //Thread callback function part
12 void cbEmptyRsv (void* pvParameters){
13  portTickType xLastWakeTime;
14  xLastWakeTime = xTaskGetTickCount();
15  for(;;){
16   //Wait for the next cycle
17   vTaskDelayUntil(&xLastWakeTime,periodEmptyRsv);
18   //Perform action here
19   //(1)Read
20   //(2)Compute
21   //(3)Write
22  }
23 }
```

Our proposed framework provides a way to add code snippets, categorized according to their functions to implement AADL components such as periodic/aperiodic threads and their interaction. In the next section, we explain our approach to design the platform-dependent code generation framework by explicitly specifying these platform-dependent software aspects.

3. PLATFORM-DEPENDENT CODE GENERATION APPROACH

Figure 2 illustrates our proposed platform-dependent framework for generating code that is executable on a heterogeneous set of target platforms. In this framework, the software behavior (which is common across all target platforms) is abstracted using a platform-independent model (PI Model). For instance, the generic behavior of an infusion pump software shown in Figure 1 can be described using an UPPAAL model. This platform-independent model is then automatically translated into platform-independent code (PI Code) using some code generator, such as TIMES [4]. The generated platform-independent code has a particular code pattern that is imposed by the generation algorithm of the code generator.

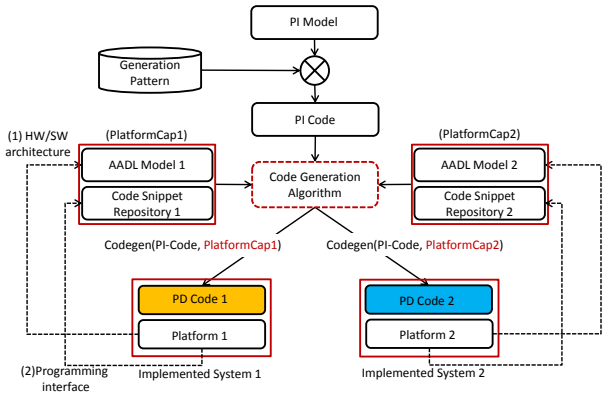

Figure 2: Platform-dependent code generation overview.

The generated platform-independent code is expected to execute on many different target platforms. To enable this, our framework explicitly specifies the platform-dependent information that is needed to execute the platform-independent code on a target platform using a *platform capability*. The platform capability captures two aspects of a platform: the hardware/software architecture and the programming interface. The hardware/software architecture is modeled using an AADL model (Figure 2-(1)), which captures interactions between the hardware and the software aspects. For example, concurrent execution of the software system is captured using a set of periodic and aperiodic thread components, whereas sensors and actuators are expressed as device components that can interact with thread components through different types of port connections. The programming interface is captured as a code snippet repository (Figure 2-(2)) that contains code snippets categorized according to their functions to implement AADL components.[1] Different platforms may fill these categories with different code snippets using their programming interfaces. Such code snippets should be distributed with the platform, e.g., as a library.

As is shown in Figure 2, the proposed code generation algorithm (denoted by the dotted box) takes the platform-independent code and the platform capabilities (AADL models and code snippet repositories) of different target platforms as inputs to generate different source code for the corresponding platforms (e.g., PD Code 1 and PD Code 2 for Platform 1 and Platform 2, respectively). In the next two sections, we show how platform capabilities can capture the platform heterogeneity using an infusion pump example.

4. AADL MODEL OF HARDWARE/SOFTWARE ARCHITECTURAL ASPECTS

We next explain how to capture architectural aspects of different target platforms as AADL models (c.f. Figure 2-(1)) using an example of two infusion pump systems. The differences in the hardware/software architectures are highlighted from the code generation perspective.

4.1 AADL components and semantics

AADL is a modeling language for describing real-time embedded systems from an architectural perspective. It provides an abstraction of software components (e.g., periodic/aperiodic threads) and hardware components (e.g., devices and processors). The interactions among such components are abstracted using ports and port connections. We note that only a subset of AADL components and their semantics is used to explain the idea underlying our framework; a broader scope of AADL components and their semantics can be found in [1].

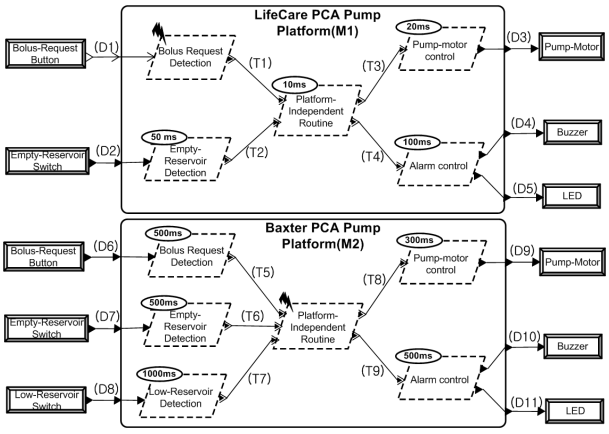

Figure 3: AADL models of two infusion pump platforms

[1]An example of the code snippet that implements a periodic thread using the FreeRTOS programming interface is shown in Listing 1.

Figure 3 shows the graphical representations of two AADL models, M1 (top model) and M2 (bottom model), that specify the hardware/software architectures of two different infusion pump systems. The big rounded box in the center of each model denotes a *system component*, which represents the scope of the software system. Each system component contains several *thread components*, represented by the dotted rectangles. Some thread components are connected to *device components*, represented by the double-lined rectangles outside the system component. Thread and device components are interconnected with each other using *port connections*, denoted by the different types of directional lines in the figure. The informal semantics of each component is as follows.

Threads: A thread is a concurrent schedulable unit of sequential computation, with one or more assigned properties. There are five threads in M1 and six threads in M2. Each thread has a property of *dispatch protocols*. For instance, the M1.BolusRequestDetection thread has an *aperiodic* dispatch protocol: any event that arrives on its input ports can invoke the thread to perform its execution; upon completion, the thread becomes idle until the next event occurrence on its input ports. In contrast, the M2.BolusRequestDetection thread has a *periodic* dispatch protocol: the thread is invoked periodically (e.g., every 500ms in the example) and independent of the occurrences of events on its input ports. There are several other properties that are not shown in the graphical format, which we will explain as is needed.

Devices: A device, such as a sensor or an actuator, is an abstraction of the physical device that exposes only its input/output ports to the external environment. For instance, the M1.BolusRequestButton and M2.EmptyReservoir-Switch in Figure 3 represent sensor-type devices that provide output ports. These output ports are connected to the input ports of the threads, M1.Bolus-RequestDetection and M2.EmptyReservoirDetection, through the port connections, D1 and D7, respectively.

Port and Port Connections: A port connection represents relationships among *ports* that enable the directional exchange of data and events. The interactions among components can be expressed using port connections. There are nine and eleven port connections (with identifiers T1–T9 and D1–D11) in M1 and M2, respectively. There are three different types of port connections (represented by different shapes in the figure), as detailed below.

- *Data* port connections express interactions between components without queuing, and a type of data message can be defined. For instance, M2.D6 is a data port connection between the device M2.BolusRequestButton and the periodic thread M2.BolusRequestDetection. The data on this connection may represent the status of the bolus request button (e.g., the button is pressed or released).
- *Event* port connections are used to deliver events among components with queuing. For example, M1.D1 is an event port connection between the device M1.Bolus-RequestButton and the aperiodic thread M2.Bolus-Request-Detection. A button-pressed event may be delivered to the aperiodic thread via queuing mechanism.
- *Event-Data* port connections are used for event transmissions with queuing, and each event may be associated with data. For example, M2.T7 is an event-data port connection between the two periodic threads, M2.Low-ReservoirDetection and M2.PlatformIndependentRoutine. The status of a low reservoir condition may be delivered via this port connection through queuing mechanism.

As was for threads, there are several port properties that are not shown in the graphical format, which we will explain as is needed.

4.2 Effects of architectural differences on the source code

We will use the two AADL models in Figure 3 to discuss the architectural differences that lead to different source code for the corresponding platforms. We assume the same platform-independent code is executed as a PlatformIndependentRoutine thread on both M1 and M2. Under this assumption, we highlight how different platforms make different choices of their hardware/software architectures to support the execution of the platform-independent code:

- Different thread types lead to different source code. For example, the thread M1.BolusRequestDetection is an aperiodic thread that is invoked only if an event is generated by the thread M1.BolusRequestButton; however, the thread M2.BolusRequestDetection is a periodic thread that is invoked every 500ms, regardless of when such an event occurs.
- Different types of port connection lead to different source code. For example, M1.T1 is an event-data port connection between two threads, M1.BolusRequestDetection and M1.PlatformIndependent-Routine. In contrast, M2.D10 is a data port connection between the M2.PumpMotorControl thread and the M2.PumpMotor device.
- Different numbers of AADL components lead to different source code. For example, M2 has LowReservoirSwitch (device), LowReservoirDetection (thread), D8 (data port connection), T7 (event-data port connection) that do not appear in M1. The more information is given in the AADL model, the more source code needs to be generated.

The above differences result in different APIs or code patterns that constitute the source code for different target platforms.

5. CAPTURING PROGRAMMING INTERFACES USING CODE SNIPPET REPOSITORIES

As was described in Section 3, the code generation algorithm takes as inputs the AADL models (described in the previous section) and the code snippet repositories that capture the platforms' programming interfaces to generate different source code that can execute on different target platforms. In this section, we explain how to capture the programming interfaces using code snippet repositories.

5.1 Code Snippet Repositories

Roughly speaking, the code generation algorithm works as follows. To generate code for a particular target platform, it first finds an AADL component (in the AADL model of the platform capability) to be generated into source code (e.g., the periodic thread of M1.AlarmControl or the event-data port connection M2.T9 in Figure 3). Then, it looks up appropriate code snippets in the corresponding code snippet repository that can implement the chosen component. Finally, it generates the source code for the platform based on these snippets. The algorithm is detailed in Section 6.

To automate the code generation process, it is necessary to construct a precise mapping between AADL components and code snippet repositories. Our approach is to construct code snippet repositories that are categorized according to the functions to implement the AADL components. Then, each category can be filled with different code snippets that are written using different programming interfaces of the target platforms. The code generation algorithm uses this categorization to find mappings between the AADL models and the code snippets.

Table 1: Categorization of code snippets

Programming support	Code snippet category	
Dispatch mechanism	Declaration Initialization Dispatch invocation function	
Periodic thread	Declaration Initialization Thread callback function	
Aperiodic thread	Declaration Initialization Thread callback function	
Device-to-Thread port connection	Data port	Declaration Initialization Get primitives
	Event and Event Data port	Declaration Initialization Interrupt callback function
Thread-to-Device port connection	Declaration Initialization Set primitives	
Thread-to-Thread port connection	Data port	Declaration Initialization (shared variables) Read primitives Write primitives
	Event and Event Data port	Declaration Initialization (FIFO queues) Read primitives Write primitives

Table 1 shows the categorization of code snippet repositories. The first column provides the code generation algorithm with the information for checking whether a target platform has the programming support to implement AADL components. The second column provides a more detailed level of categorization that guides code snippets of each programming support to be written in a particular format. For example, consider the periodic thread component M2.EmptyReservoirDetection in Figure 3. To generate the code that is mapped to this periodic thread component, the code generation algorithm refers to the periodic thread category in the programming support column in Table 1 to check whether code snippets of each subcategory of the programming support exist, and if so, it uses the corresponding code snippets to generate the code.

5.2 Case Study: FreeRTOS vs. bare platform

We now demonstrate the applicability of the categorization of code snippet repositories in Table 1 using a case study of two different platforms, denoted by *FreeRTOS* and *BarePlatform*, which have different programming interfaces. The *FreeRTOS* platform runs the FreeRTOS operating system, which supports a preemptive scheduler with which programmers implement periodic/aperiodic threads. The *BarePlatform* platform is a bare platform that does not run any operating system; therefore, one needs to implement a dispatch mechanism that can invoke periodic/aperiodic threads. We explain each category of the categorization shown in Table 1 using example code snippets implemented on these two platforms.

Dispatch mechanism: A programming interface should provide a dispatch mechanism, i.e., periodic or aperiodic, for threads to be scheduled. Some platforms may already have such a dispatch mechanism implemented; for example, *FreeRTOS* provides the API vTaskStartScheduler(), and it is sufficient to call this function to start the dispatch mechanism. However, other platforms (e.g., *BarePlatform*) may not have a dispatch mechanism, in which case the platforms should add code snippets that implement dispatch mechanisms following the Dispatch mechanism category in Table 1.

Listing 2: Code snippets of Dispatch mechanism on the *BarePlatform*

```
1  //Declaration
2  const int Dispatch_Invocation_Interval = 10;
3
4  //Initialization
5  void init_dispatch( void ){
6   hardware_timer_init(Dispatch_Invocation_Interval,
        cbDispatchInvocation);
7  }
8
9  //Dispatch invocation
10 void cbDispatchInvocation (void* pvParameters){
11  Disable_interrupt;
12    Update_Dispatch_Flag();
13    Dispatch_Aperiodic_Threads();
14    Dispatch_Periodic_Threads();
15  Enable_interrupt;
16 }
```

Listing 2 gives an example code snippet of the dispatch mechanism implemented on *BarePlatform*, which belongs to the code snippet category of Dispatch mechanism in Table 1. The operation of the dispatch mechanism is as follows. In Lines 4–7, the code snippet initializes a hardware timer of the microprocessor with a fixed millisecond-basis period (Dispatch_Invocation_Interval) defined in Line 2 of the *Declaration* part, and a pointer to the callback function (cbDispatchInvocation) in the *Initialization* part. This enables cbDispatchInvocation, implemented in the *Dispatch invocation* part, to be called every period (i.e., 10 ms in this example). Upon being activated, the invocation function checks the list of periodic and aperiodic threads[2] that need to be invoked at the current invocation period; this is implied in Update_Dispatch_Flag() in Line 12. Then, the invocation function executes all checked threads in Lines 13–14, and it completes the current dispatch round.

Periodic/Aperiodic thread implementation: A programming interface should provide a mechanism to implement periodic and aperiodic threads that can be scheduled by the dispatch mechanism explained above. Code snippets that implement such threads vary across platforms that expose different programming interfaces.

Listings 3 and 4 show the code snippets for aperiodic threads that can be executed on *FreeRTOS* and *BarePlatform*, respectively. Special functions (e.g., F_{tid}(ext)) enclosed with two sharp signs (#) are used to specify parametrized code snippets that need to be replaced some other codes, which we will explain in the next subsection. Both code snippets implement aperiodic threads on each platform; however, they are different in the following sense.

Listing 3: Parametrized code snippet of Aperiodic thread in FreeRTOS

```
1  //Declaration part
2  const portBASE_TYPE prioirty#F_tid(ext)#=#F_priority(ext)#;
3  const portBASE_TYPE stacksize#F_tid(ext)#=#F_stack(ext)#;
4
5  //Initialization part
6  void init_#F_tid(ext)#( void ){
7   TaskCreate(cb#F_tid(ext)#,''#F_tid(ext)#'',stacksize#F_tid(
        ext)#, NULL,prioirty#F_tid(ext)#,NULL);
8  }
9
10 //Thread callback function part
11 void cb#F_tid(ext)# (void* pvParameters){
12  for(;;){
13    #F_input(ext)# //Wait for the input
14    //Compute
15    #F_output(ext)# //Write output
16  }
17 }
```

[2]This information is implemented in the declaration part, but not shown here for clarity

Listing 4: Parametrized code snippet of Aperiodic thread in BarePlatform

```
1  //Declaration part
2  const int prioirty#F_tid(ext)#=#F_priority(ext)#;
3
4  //Initialization part
5  void aperiodic_#F_tid(ext)#_init( void ){
6    register_aTask(cb#F_tid(ext)#, prioirty#F_tid(ext)#);
7  }
8
9  //Thread callback function part
10 void cb#F_tid(ext)# (void* pvParameters){
11    #F_input(ext)# //Read input
12    //Compute
13    #F_output(ext)# //Write output
14 }
```

The two programming interfaces require different information to initialize aperiodic threads. For example, the *FreeRTOS* code snippet in Listing 3 specifies the stack size (Line 3) of the maximum amount of memory that a thread can occupy at run time. This parameter is passed to the *TaskCreate* API (Line 7), so that the OS kernel triggers exceptions in case of stack overflows at run time. On the contrary, *BarePlatform* does not require explicit stack sizes of threads, since the platform is incapable of specifying stack sizes and capturing stack overflows.

The two programming interfaces provide different ways of implementing interactions between their thread callback functions and dispatch mechanisms. For example, the *FreeRTOS* code snippet in Listing 3 implements the infinite for-loop (Line 12–16), in which blocking functions are used to interact with the scheduler (the function #F_{input}(ext)# in Line 13 will be replaced with such blocking functions in our framework). However, the *BarePlatform* code snippet in Listing 4 does not have such a for-loop: its dispatch mechanism is invoked periodically, and the called thread callback function will be returned in the current dispatch invocation without looping.

The two programming interfaces provide different names of APIs to perform similar functions. For instance, to register the thread information to the kernel, the *FreeRTOS* uses the *TaskCreate* API (Line 7), whereas the *BarePlatform* uses the *register_aTask* API (Line 6).

Note that the code snippet category of aperiodic threads in Table 1 is filled with different code snippets of Listing 3 and Listing 4 for *FreeRTOS* and *BarePlatform*, respectively.

Port connection implementation: Each programming interface should provide a mechanism to implement the port connections that enable thread and device components to interact with one another (e.g., D1–D11 and T1–T7 in Figure 3). Different types of port connections (e.g, data or event-data ports) lead to different code snippets. Moreover, different instances of a port connection can be implemented by different code snippets, depending on whether they cut through the software system scope (that need to be generated into source code) or not. For example, M1 and M2 in Figure 3 contain system components (big rounded boxes in the middle) that represent the scope of the software system; therefore, thread components within the scope are subject to code generation. In contrast, device components outside the scope are not subject to code generation. Finally, the port connections of M1.{D1-D5} and M2.{D6-D11} cut through the scope of software system. In our code snippet repository, such port connections are separately categorized from port connections that have both of their source and destination components residing inside the scope (e.g., M1.{T1-T4} and M2.{T5-T9})

Based on the above observation, the code snippet repository in Table 1 distinguishes three different categories of port connections:

- The *Device-to-Thread* port connection category stores code snippets that are used to implement directional port connections from *device* to *thread* components. This port connection is typically used by thread components to read sensor values. For example, the implementations of M1.{D1, D2}, M2.{D6, D7, D8} use code snippets from this category. This category includes code snippets that process *input only* (i.e., it does not have a code snippet that writes outputs). Inputs can be read from devices in two different ways. First, a thread component may read values from a device component through data ports (e.g, M1.D2 and M2.{D6, D7, D8}); in this case, the code snippet provides *Get* primitives that can be called by thread components to retrieve data from device components. Second, a thread component may read values from device components through event or event-data ports (e.g., M1.D1); in this case, the code snippet provides the *Interrupt* callback function that is called when events from device components occur.
- The *Thread-to-Device* port connection category stores code snippets that are used to implement directional port connections from *thread* to *device* components. This port connection is typically used by thread components to actuate actuators by writing values. For example, the implementations of M1.{D3, D4, D5}, M2.{D9, D10, D11} use code snippets from this category. This category includes code snippets that process *output only* (i.e., it does not have a code snippet that reads inputs). Thread components can write outputs to devices by calling the *Set* function.
- The *Thread-to-Thread* port connection category stores code snippets that are used to implement port connections between two thread components. For example, the implementations of T1-T9 in M1 and M2 use code snippets from this category. Unlike the above port connection types, this category includes code snippets that process *both* inputs and outputs. Two threads can communicate with each other through either data port or event-data port. In the former case, a shared variable is used with the associated read/write primitives; in the latter, a FIFO queue is used with the associated read/write primitives.

We next give an example of the code snippet for thread-to-thread port connection implementation on *FreeRTOS*. (Due to space constraints, we cannot show examples of each port connection category for both platforms.)

Listing 5 shows the code snippet of (event-data port) thread-to-thread port connection of *FreeRTOS*. We note that this code snippet is written following the four categories related to the (event-data port) thread-to-thread port connection shown in Table 1.

Declaration: Lines 1–6 implement the declaration part, which lists the necessary variables to implement the port connection using a FIFO queue: the handler of the FIFO queue (Line 2), the queue size (Line 3), the dequeue policy (Line 4), the blocking mode (Line 5), and the overflow handling policy (Line 6).

Initialization: Lines 8–11 implement the initialization function that creates a FIFO queue using FreeRTOS API, *xQueueCreate*; the variables in the declaration part is passed to the API call.

Read primitive: Lines 13–28 implement the read primitives that can be used by threads to read items from the FIFO queue. Note that two different read primitives are implemented, and which one is generated depends on the dequeue policy that is specified in Line 4. This is intended to capture the AADL property *Dequeue_Protocol*, whose value can be either *OneItem* (read a single item from the queue) or *AllItems* (read all items from the queue).

xQueueReceive is a FreeRTOS API that dequeues items from the queue. In case the queue is empty, the blocking mode, specified as edQBlockMode#F_{pid}(ext)# in Line 5, decides whether a caller of this API should be blocked until any item arrives, or it should be timed-blocked (i.e., blocked for only a certain amount of time), or non-blocked (i.e., never blocked). The *uxQueueMessagesWaiting* API in Line 24 returns the number of items in the queue, which is needed to read all the items from the queue (Lines 23–25).

Write primitive: Lines 30–49 implement the write primitive that can be called by threads to insert items to the FIFO queue. The *xQueueSend* API is used to insert an item to the FIFO queue in Line 32. Lines 34–45 show the three different ways for handling the queue overflow exception, depending on the value of the AADL property *Overflow_Handling_Protocol* (i.e., DROP_OLDEST or DROP_NEWEST or ERROR). The code that handles such an overflow handling protocol is not detailed here.

Listing 5: Parametrized code snippet of Thread-to-Thread port connection in FreeRTOS

```
1  //Declaration part
2  static xQueueHandle edQHandle#F_pid(ext)#;
3  const portBASE_TYPE edQSize#F_pid(ext)# = #F_qsize(ext)#;
4  const portBASE_TYPE edDQPolicy#F_pid(ext)# = #F_rpolicy(ext)
       #;
5  const portBASE_TYPE edQBlockMode#F_pid(ext)# = #F_wmode(
       ext)#;
6  const portBASE_TYPE edQOverflowHandling#F_pid(ext)# = #
       F_wpolicy(ext)#;
7
8  //Initialization part
9  void ed#F_pid(ext)#_Init( void ){
10   edQHandle#F_pid(ext)# = xQueueCreate(edQSize#F_pid(ext)#,
       sizeof(#F_itemtype(ext)#));
11 }
12
13 //Read primitive
14 #if edDQPolicy#F_pid# == OneItem
15 portBASE_TYPE Read_#F_pid(ext)# (#F_itemtype(ext)#* buf){
16   xQueueReceive(edQHandle#F_pid(ext)#, buf, edQBlockMode#
       F_pid(ext)#);
17   return TRUE;
18 }
19 #elif edDQPolicy#F_pid# == AllItem
20 portBASE_TYPE Read_#F_pid# (#F_itemtype(ext)#* buf){
21   portBASE_TYPE item_count = 0;
22   item_count = uxQueueMessagesWaiting(edQHandle#F_pid(ext)#
       );
23   for(int i = 0 ; i < item_count ; i++){
24     xQueueReceive(edQHandle#F_pid(ext)#, buf + i*sizeof(#
       F_itemtype(ext)#), edQBlockMode#F_pid(ext)#);
25   }
26   return item_count;
27 }
28 #endif
29
30 //Write primitive
31 portBASE_TYPE Write_#F_pid(ext)# (#F_itemtype(ext)#* buf){
32   portBASE_TYPE result = xQueueSend(edQHandle#F_pid(ext)#,
       buf, edQBlockMode#F_pid(ext)# );
33   if(result == FALSE){
34     switch(edQOverflowHandling#F_pid(ext)#){
35     case DROP_OLDEST:
36         //Drop oldest and enqueue
37         break;
38     case DROP_NEWEST:
39         //Drop newest and enqueue
40         break;
41     case ERROR:
42         //Raise an exception
43         break;
44     default:
45     }
46     return FALSE;
47   }
48   return TRUE;
49 }
```

5.3 Parametrized code snippets

As described above, the AADL models and the code snippet repositories are created independently of one another. Hence, it is necessary to inform the code generation algorithm of the scope of the code that should be related to the information of the AADL model of the target platform. This is done via parametrized code snippets, which specify the placeholders that later can be filled by the code generation algorithm based on the AADL model. Therefore, our framework separates concerns between how the code snippets are written and how they are actually used in a certain architectural context. For example, the code snippet for aperiodic threads in Listing 3 is written independently of how it is used to implement M1.BolusRequestDetection or M2.PlatformIndependentRoutine in Figure 3.

Parametrized code snippets can be written using functions that take a set of input parameters and return a piece of code. Such functions enable a parametrized code snippet to be instantiated into several different pieces of code; for instance, the code snippets in Listing 3, 4, 5 use the parametrized code snippets that are enclosed with two sharp signs(#). These functions also specify the rules for instantiating the code using external information that is passed through *ext*. For example, F_{tid}(ext) in Line 11 of Listing 3 specifies how a thread identification should be represented in the aperiodic thread callback function using external information *ext*. Suppose the string "BolusRequest" is passed as *ext*, then one may define a function F_{tid} that converts the string into "BolusReq" and returns it as a piece of code. Then, the code snippet after resolving the parameter of Line 11 becomes "*void cbBolusReq (void* pvParameters)*."

As another example, one may define a function, F_{period}(ext), to specify the rule to convert *ext* into a value that represents the period of a periodic thread in the code snippet. In Figure 3, the period property of M2.EmptyReservoirDetection is represented as a string of "500 ms". This string is passed as a parameter of *ext*. The internal of F_{period} converts "500 ms" into some appropriate values. Suppose, we want to get the code in Line 2 of Listing 1. Then, the function simply removes "ms" from the string, resulting in the value 500. However, the returned value is not always necessarily 500 to represent "500 ms" of period in the code snippet. Because different target programming interfaces may require to scale the numeric value differently to represent "500 ms" of period. For example, the dispatch mechanism of BarePlatform in Listing 2 invokes the dispatch mechanism every 10 ms as specified in line 2. The code snippet for periodic threads (not shown here) requires to specify periods of thread relative to the dispatch invocation interval. Then, F_{period}(ext) should convert "500 ms" into 50 instead of 500. This example shows different platforms interpret the same information coming from the AADL model in different ways. Therefore, having parametrized code snippets provides flexibility to deal with such heterogeneous programming interfaces.

In the next section, we explain the proposed code generation algorithm that co-relates an AADL model (explained in Section 4) and a code snippet repository (explained in Section 5), in order to produce the platform-dependent code for different platforms.

6. PLATFORM-DEPENDENT CODE GENERATION ALGORITHM

The constructed AADL model and the code snippet repository of a particular target platform are used together with the platform-independent code as inputs to the code generation algorithm to generate the platform-dependent code that is executable on the target platform (c.f. Figure 2). In this section, we highlight how

the algorithm processes and correlates the AADL model and code snippet repository of the platform; we discuss how the platform-independent code can be composed in Section 7.

The AADL model is expressed in textual form. Listing 6 shows a textual representation of M2.EmptyReservoir-Detection in Figure 3. The scope starting with **thread** (Line 2) and ending with **end** (Line 15) characterizes M2.Empty-ReservoirDetection. Lines 4–9 characterize the input and output ports that are associated with the thread. Lines 11–14 specify the properties that characterize the periodic thread.

Listing 6: Textual representation of the periodic thread component M2.EmptyReservoirDetection in Figure 3

```
1  -- Define thread type of EmptyReservoirDetection
2  thread thd_empty_rsv
3    features
4      D7: in data port;
5      T6: out event data port{
6        Overflow_Handling_Protocol => Error;
7        Dequeue_Protocol => AllItems;
8        Queue_Size => 5;
9      };
10   properties
11     Dispatch_Protocol => Periodic;
12     Period => 500 Ms;
13     SEI::Priority => 2;
14     Source_Stack_Size => 500 B;
15 end thd_empty_rsv
```

We note that the AADL standard provides a rich set of properties to characterize AADL components. In addition, one may also define custom properties associated with AADL components. As a result, it is difficult (and out of scope of this paper) to design a code generation algorithm that takes into account all possible properties. Instead, we provide a finite set of AADL properties that are sufficient for our case study; we expect that extensions of this property set can easily be done as are needed.

Table 2 summarizes the list of AADL properties of thread and port connection components that are used by our code generation algorithm in order to find appropriate code snippets in the code snippet repository.

Table 2: Information extracted from the AADL model

AADL component	Property	Example value
Thread	Thread ID	EmptyReservoirDetection
	Thread type	Periodic/Aperiodc
	Thread period	100ms or 10sec
	Thread priority	3
	Source stack size	500 B
	Input port connection IDs	D1, D2, T1
	Output port connection IDs	D1, D2, T1
Port connection	Connection ID	D1, D2, T1
	Interaction type	Device-to-Thread, Thread-to-Thread
	Source component ID	EmptyReservoirDetection
	Destination component ID	PlatformIndependent-Routine
	Port connection type	Data, Event, Event-Data
	Queue size	5
	Read policy	Read one item, Read all item
	Write policy	Drop oldest, Drop newest, Error

Listing 7 gives the pseudo code of the code generation algorithm that generates platform-dependent code from an AADL model and a code snippet repository. The parameters M and C that are passed as input to the function *CodeGen* in Line 16 are the abstracted representation of the information obtained from the AADL model in Table 2 and the code snippet repository in Table 1, respectively.

Listing 7: Pseudo code of the platform-dependent code generation algorithm

```
16 function CodeGen(M, C)
17 exp_scope
18   //Generating code for thread components
19   for each thread, Thread[i] ∈ M
20     if M.Thread[i].type == PERIODIC
21       SnippetHandle := C.PeriodicThreadSnippet;
22     else if M.Thread[i].type == APERIODIC
23       SnippetHandle := C.APeriodicThreadSnippet;
24     endif
25     for each parametrized function, F_k ∈ SnippetHandle
26       SnippetHandle.F_k(M.Thread[i]);
27     endfor
28     Generate(SnippetHandle);
29   endfor
30  //Generating code for port connection components
31   for each port connection, PortConn[j] ∈ M
32     if M.PortConn[j].type == Device-to-Thread
33       SnippetHandle := C.Device-to-ThreadSnippet
34     else if M.PortConn[j].type == Thread-to-Device
35       SnippetHandle := C.Thread-to-DeviceSnippet
36     else if M.PortConn[j].type == Thread-to-Thread
37       SnippetHandle := C.Thread-to-ThreadSnippet
38     endif
39     for each parametrized function, F_l ∈ SnippetHandle
40       SnippetHandle.F_l(M.Thread[i]);
41     endfor
42     Generate(SnippetHandle);
43   endfor
44 exception(No matched code snippets)
45   //Exception handling
46 exception(No matched parameters)
47   //Exception handling
48 endexp_scope
49 endfunction
```

The algorithm generates the platform-dependent code for thread components of M in Lines 19–29. The for-loop in these lines finds a match between thread components in M and the code snippets in C. Using the *dispatch protocol* property of M, the algorithm finds different code snippets from C. After such a match is found, the algorithm resolves the parametrized code snippets (explained in Section 5.3) of the matched code snippet in Lines 25–27. The properties of the matched thread in M are passed as a parameter to the function of the parametrized snippets. This function converts the parameter into a piece of code using the rules specified in the function. After resolving all parametrized code snippets, the algorithm generates the platform-dependent code of the thread components in M, which is implied in Line 28; here, *Generate* is a simple function that copies and pastes the code snippet into some output files. The code generation for port connection components (Lines 31–43) is similar to the generation for thread components, except that the generation algorithm uses the *interaction type* property of the port connection to find a match between M and C.

We note that the code generation algorithm deals with two types of exceptions implied in the exception scope (Lines 17–48). Specifically, the *No matched code snippets* exception is raised when the algorithm cannot find a matched code snippet in C to generate a component in M. For example, the code snippet repository may not contain code snippets that implement periodic threads, but the AADL model has periodic thread components to be generated into code. One should handle such an exception appropriately, e.g., by registering a code snippet to C that implements periodic threads. The *No matched parameters* exception is raised when the algorithm cannot find a match of a conversion function in any parametrized code snippets. There are two cases to trigger this exception: (1) M does not have the information that is needed for C to gener-

ate code; for example, C requires the blocking mode on the FIFO queue (#F_{wmode}(ext)#) to be specified, so as to implement read-/write primitives for port connection components in Line 5 of Listing 5, but M does not have such a property in Table 2, and (2) C does not have the capability to implement the properties of M; for example, M has a property of source stack size, but C does not have the code snippet that contains this information. Both exceptions should be handled by users (e.g., by adding missing information in M or by using the default value of C).

7. CASE STUDY

This section presents a case study of infusion pump systems. We first explain our experimental platforms that run the source code generated from the proposed code generation framework. Then, we discuss several considerations when the platform-independent code needs to be integrated as a part of the platform-dependent code.

Experimental platforms: Patient-Controled Analgesia (PCA) infusion pump systems are safety-critical medical devices that are used to inject drugs to the patients for pain-relief. A special button is attached to the pump with which patients may request additional drug, called a bolus, by pressing the button. As a case study, we have extended the Generic PCA (GPCA) testbed [8] to apply the proposed framework to generate source code for two different infusion pump systems.

The hardware and software architectures of both PCA pump systems are captured using different AADL models, M1 and M2, in Figure 3. M1 and M2 express the hardware and software architecture of Lifecare PCA pump (right) and Baxter PCA pump (left) in Figure 4 respectively. The differences of the two target platforms are summarized in Section 4.2.

Figure 4: Evaluation platform: PCA infusion pump systems

We connected the sensors and actuators of each PCA infusion pump to different microcontrollers. The sensors and actuators of Lifecare pump are interfaced using HCS12 microcontroller, and those of Baxter pump are interfaced using ARM7 microcontroller. The software stack that each microcontroller operates is also different. The microcontroller of Lifecare pump does not run a full software stack, such as a real-time operating system. Therefore, the generated code needs to include an implementation of dispatch mechanisms, in addition to using the interface provided by the dispatch mechanism within the thread code. An example of such a dispatch mechanism is shown in Listing 2. On the other hand, the microcontroller of the Baxter pump is running FreeRTOS. This means we can rely on the programming interface provided by the OS kernel to implement periodic/aperiodic threads. In the preceding sections, we have shown how our code generation framework captures this platform heterogeneity in AADL models and code snippet repositories.

Composition with Platform-Independent Code: As illustrated in Figure 2, the platform-independent code (PI Code) generated from the platform-independent model (PI Model, e.g, Figure 1) should be composed with different platform capabilities (PlatformCap1 and PlatformCap2) so that the generated platform-dependent codes (PD Code 1 and PD Code 2) implement the same software behavior expressed as a platform-independent model on different target platform (Platform 1 and Platform 2).

The composition of the platform-independent code and the platform capability is out of scope in this paper. However, we explain several places that need to be considered in our framework for the composition.

Figure 1 is the example of the platform-independent model that expresses the infusion pump software behavior using a state-transition system with several inputs and outputs. Such a state-transition system can be systematically transformed into source code that repeats the following sequential operations, which is also used in several code generators [2][4] :

1. *Read* inputs from some variables, PI_{input}, that is updated by some external piece of code
2. *Compute* the next state using transition tables (encoded as switch-case statements or array structures) based on PI_{input}
3. *Write* outputs to some variables, PI_{output}, that are read by some external piece of code

In our framework, the AADL model utilizes port connections to express input/output relationship among different AADL components such as threads and devices. On the other hand, the code snippet repository in Table 1 contains code snippets of the port connection that implement read and write primitives using the target programming interfaces. An example of read/write primitives can be found in Listing 5. In order to compose the platform-independent code and the platform capability, one needs to resolve (1) read dependencies between PI_{input} and the read primitives, and (2) write dependencies between PI_{output} and the write primitives. In Listing 4, #F_{input}(ext)# (line 11) and #F_{output} (ext)# (line 13) specify such placeholders in the form of parametrized code snippets (i.e., one should provide the implementations of such functions to resolve input/output dependencies).

In addition, the semantics of *repeated* execution of (1), (2), (3) in the platform-independent code needs to be mapped to the platform capability. A thread component of the AADL model is an abstraction of sequential computation in which input is read from input port connections and output is written to output port connections. Therefore, one may implement such a mapping using either periodic or aperiodic thread components. In case of periodic threads, the execution of (1), (2) and (3) can be performed periodically. In case of aperiodic threads, the execution of (1), (2) and (3) can be performed only if any input is available from one of the input ports. Either case equally implements the platform-independent model by executing (1) (2) (3) repeatedly.

8. RELATED WORK

Some works studied separating concerns between specification and hardware-dependent details in generating source code. [12] proposes automatic generation of hardware dependent software for

MPSoCs platforms from abstract system specifications. The approach of this work is similar to ours in the sense that one can write specifications of embedded systems hiding the details of implementation, and later mapping to an actual platform to generate code is done separately. However, this work uses different modeling language, Transaction Level Models (TLM), as an input to the process as opposed to AADL that we use. In addition, they do not consider dealing with heterogeneous aspects of programming interfaces as our parametrized code snippets.

There are several works related to code generation from AADL models. [9] presents the OCARINA tool-suite that allows automatic code generation from AADL models. However, the code generation algorithm used in this tool targets a single platform, POLYORB and POLYORB-HI, by hard-wiring the programming interface and code patterns of the platform inside the generation algorithm. In contrast, our approach parameterizes the code generation with the platform description to provide flexibility in choosing different target platforms. [6] raises an open question about the need of flexibility in code generation through an experimentation that generate C code (compliant with OSEK/VDX) and OIL configuration code from the AADL model. Our work can be one approach to achieve such flexibility in the code generation process.

AADL runtime services defined in the AADL standard provide primitives that can be used by application source code. For example, the *Send_Output* and *Receive_Input* runtime services allow threads to exchange data through ports, which is similar to the read-/write primitives that our code snippet repository provides. Application code running on different platforms may use these services for communication if those platforms operate the AADL runtime. However, implementing an AADL runtime that supports a wide range of embedded platforms is difficult in practice. Moreover, operating AADL runtime is sometimes overkill for some embedded systems that only require simple communication mechanisms through shared variables or FIFO queues among threads without having any network communication. Such mechanisms may be implemented more efficiently if one directly uses the APIs provided from the underlying software layer, e.g., an RTOS.

Platform heterogeneity has also been considered in existing research on UML. For instance, [7] observed that the underlying zero execution time semantics of the UML State Machines makes it hard to achieve semantically correct implementations, and it proposed a syntactic extension of the UML State Machines and an accompanying semantics that can be used to describe platform-independent model, as well as an approach for synthesizing PSM (Platform-Specific Model) and code. This work differs from ours in that it focused on platform-specific scheduling aspects instead of the programming interfaces; at the same time, our work complements the work in [7], since both address different important issues of the platform heterogeneity.

Reducing error-prone process during system integration is also studied from the interface synthesis field. [11] proposed a general interface synthesis flow that can be applied for different applications. The proposed framework separates software generation from interface generation. [13] proposed a model-based framework that enables to specify software component interaction and the mapping between Architecture Description (AD) models and platform-specific API calls. However, these works do not solve the problem of programming interface heterogeneity that our work focuses on.

9. CONCLUSION

We proposed a platform-dependent code generation framework that generates different source code for different target platforms. In our framework, platform heterogeneity is characterized using platform capabilities that capture different hardware/software architectures and programing interfaces. The AADL models are used to express the hardware/software architectures of target platforms, using thread, device, and port connection components. The code snippet repositories contain code snippets that are categorized according to the functions to implement the AADL components. Our code generation algorithm generates different source code for different target platforms from the corresponding AADL models and code snippet repositories.

We envision the proposed framework as a fully automatic technology, as was illustrated in our infusion pump case study. To ensure that the automation can be applied to a wide range of target platforms, we plan to conduct an extensive set of case studies to generate platform-dependent code on a broad range of common embedded platforms. We also plan to study the generation patterns of existing code generation tools for platform-independent code so that our code generation algorithm leverages such patterns to automatically integrate the platform independent and dependent code.

10. ACKNOWLEDGMENTS

This research was supported in part by NSF grants CNS-1035715, CNS-1042829, and CNS-1117185.

11. REFERENCES

[1] Architecture Analysis and Design Language. http://www.aadl.info.

[2] Simulink Coder: Generate C and C++ code from simulink and stateflow models. http://www.mathworks.com/products/simulink-coder.

[3] Using the freertos real-time kernel. http://www.freertos.org.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, 2003.

[5] G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185 of *LNCS*, pages 200–237, 2004.

[6] M. Brun, J. Delatour, and Y. Trinquet. Code generation from aadl to a real-time operating system: An experimentation feedback on the use of model transformation. In *ICECCS*, 2008.

[7] S. Burmester, H. Giese, and W. Schafer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Model Driven Architecture – Foundations and Applications*, volume 3748 of *LNCS*, pages 25–40. 2005.

[8] B. Kim, A. Ayoub, O. Sokolsky, I. Lee, P. Jones, Y. Zhang, and R. Jetley. Safety-assured development of the gpca infusion pump software. In *EMSOFT*, 2011.

[9] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In *Ada-Europe*, 2009.

[10] T. MathWorks. Stateflow: For state diagram modeling.

[11] A. Rajawat, M. Balakrishnan, and A. Kumar. In *VLSI Design*, 2000.

[12] G. Schirner, A. Gerstlauer, and R. Dömer. Automatic generation of hardware dependent software for MPSoCs from abstract system specifications. In *ASPDAC*, 2008.

[13] G. Waignier, P. Sriplakich, A.-F. Meur, and L. Duchien. A model-based framework for statically and dynamically checking component interactions. In *MoDELS*, 2008.