This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

Prototyping Wearables: a Code-First Approach to Designing Embedded Systems

Daniel Graham, Gang Zhou Senior Member IEEE

***Abstract*—As wearable devices become ubiquitous, there will be an increased demand for platforms that allow engineers and researchers to quickly prototype and evaluate new wearable devices. This paper presents a platform that allows researchers and engineers to quickly prototype new wearable devices using a code-first approach. This approach allows software developers to create new prototypes by first writing the code that the prototype is required to run. Once the code has been written, the hardware that is required to run the application can be generated by analyzing the code that the software developer has specified. This code-first approach is possible because of the system's architecture which is comprised of both a hardware component and a software component. The hardware component consists of a main board with four types of expansion ports, while the software platform is a modular middleware, which consists of a collection of stateless libraries that abstract each hardware module. These modular abstractions allow us to synthesize the hardware configuration from the software definition. We evaluated our design by using it to prototype three wearable devices: an environmental exposure monitoring smartwatch, an infrared indoor localization system and a step counter.**

***Keywords*—Wearable computing, Hardware, Firmware**

I. Introduction

In 1988, the chief scientist at Xerox PARC, Mark D. Weiser, coined the term ubiquitous computing. He envisioned a future of ubiquitous computing that he called:"Calm Computing". He believed that computers should create calm by being quiet and invisible servants that help us to be more efficient in a way that feels intuitive [39]. His vision has inspired several new computing devices, including wearable devices.

These wearable devices have changed the way we perceive personal computing devices. Devices such as the Galaxy Gear [3], the Pebble [2] and the Fitbit [1] have created new ways for us to track our health and check our mail. However, as researchers and engineers begin to explore the potential of wearable devices, they are faced with the challenge of developing and testing custom hardware prototypes.

As we begin to consider the question of prototyping devices by generating hardware from code, there are two fundamental contexts in which the question should be considered. The first context is automatically generating hardware prototypes by analyzing the code they are required to run, thereby allowing software developers with limited hardware experience to develop their own prototypes. The second context, is that of hardware/software co-design, where the experienced hardware engineer is interested in optimizing a hardware design by utilizing information from the code that the platform is required to run. Thanks to research done by several researchers, we

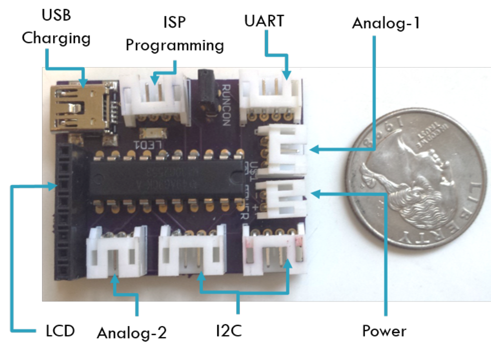


Fig. 1. Shows the layout of the mainboard (E-unit)

know a great deal about problems related to hardware/software co-design [37] [16] [14] [11] [12] [36].

In this paper, we consider the first context and attempt to reduce the time that it takes to develop a hardware prototype by proposing a code-first approach to the design of embedded systems. A code-first approach allows a software developer to begin developing a hardware prototype by first writing the code that it will run. After the code has been written, it can be analyzed to determine the hardware configuration that is required. Once the configuration has been determined, a list of modules and their appropriate ports are displayed and the system can be configured by plugging in the appropriate modules. After the software developer has tested the code on the configured hardware prototype, he or she can then automatically generate the design files that are required to fabricate a custom board. This is possible because the platform is comprised of a collection of modular software and hardware components. The hardware components consist of a main board with several hardware modules, while the software component is a modular middleware which consists of stateless libraries, which abstract each hardware component. This modular abstraction creates a direct relationship between the software module and the hardware module, thereby allowing us to synthesize a hardware configuration from a software definition. A code-first approach to designing embedded systems can be achieved by creating a direct mapping between a stateless modular middleware and modular hardware.

The intuition behind the board's design is that there are a collection of interfaces (i.e. SPI, I^2C, and UART interfaces) that components normally use to interface with microprocessors. By abstracting these interface communication protocols and directly exposing the associated pins, it is possible to design a board that allows software developers to automatically generate the hardware configurations that are required to run their software.

Figure 1 shows a picture of the main board, which we call the E-unit. The E-unit is comprised of four expansion ports. Each port is designed to be compatible with a specific interface and therefore accommodates a particular type of module. The first port is designed to be compatible with I^2C interfaces and accommodates modules (such as an accelerometer and gyroscope). The second port is designed to interface with UART-based modules (such as a Bluetooth module). The third port is designed to interface with analog sensing modules (such as a humidity sensor or finger pulse sensor). The fourth and final port is an SPI port that is designed to be compatible with a display module.

In the past, hardware platforms have helped to catalyze innovation [19] [32]. In 2005, researchers at the University of California Berkeley released the Telos mote along with the TinyOS operating system. This platform provided computer scientists with the tools to design and evaluate new protocols for wireless sensor networks and devices. Unlike the Telos platform, our proposed platform is reconfigurable. This means that components can be added or removed from the platform. Reconfigurable platforms for prototyping large-scale devices are becoming increasingly popular.

This paper makes four contributions:

- it presents a platform for prototyping wearable computing devices;
- it proposes a code-first approach to the design of embedded systems that allow programmers to synthesize hardware configurations from software definitions;
- it presents a method for automatically generating custom schematics from a hardware configuration; and
- it presents an approach for automatically identifying port conflicts in hardware/software co-designs.

II. Software Design

This section is divided into two subsections. In the first subsection, we present our proposal for a code-first approach to developing embedded systems. In the second subsection, we discuss the design of the modular middleware architecture that is used in our reconfigurable platform.

A. A Code-First Approach to Embedded System Design

A code-first approach allows software developers to write an application without worrying about configuring its resource dependencies. For example, Microsoft's entity framework allows software developers to abstract database models as classes [9]. Once the user compiles the program, the framework will automatically generate the database's structure from the class definitions. This increases the programmer's productivity since she does not need to manage the database by writing SQL queries to create, update and delete tables. Instead, the entity framework ensures that the database is compatible with the programmer's implementation.

A code-first approach to embedded system design allows the framework to configure the hardware platform by analyzing the code for hardware dependencies. These dependencies are then used to generate the hardware configuration that is needed to run the application. This is possible because of the modular architecture that maps one software module directly to one hardware component. This one-to-one mapping allows hardware dependencies to be determined by analyzing the software dependencies. However, it is not sufficient to only analyze software dependencies, the microcontroller may have physical constraints that may prevent a valid software model from being executed. For example, the software developer may include two UART libraries in their program. Though this is a valid software model some microcontrollers such as the MPG430G2553 can only accommodate a single UART module without the use of resource sharing hardware. To ensure that our approach only produces valid hardware configurations we need to verify that the software models are compatible with our hardware platform and microcontroller. In the following sections, we discuss the design of the modular architecture and demonstrate how it can be used to construct the software models.

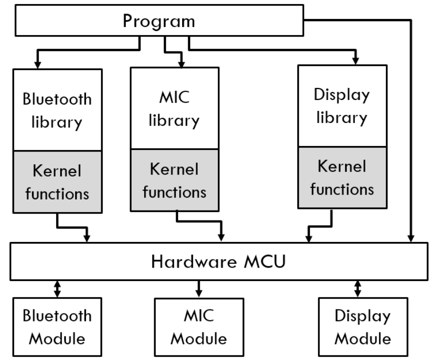


Fig. 2. An overview of the middleware architecture

B. Modular Middleware Architecture for Embedded Systems

Developing embedded software is tedious. The software developer needs to know what control registers to set and what data registers to read. This means that the developer needs to have intimate knowledge of the chip's architecture and the board's layout, and therefore must spend hours reading datasheets. In an effort to reduce the burden on the software developer, several researchers have proposed a collection of hardware abstractions to help address this problem [17] [18]. We take a similar approach by proposing a modular middleware architecture that allows the software developer to quickly build software for our platform. Figure 2 shows an overview of the modular middleware architecture. The middleware is comprised of a collection of libraries/software modules. Unlike previous systems that associate modules with generic functionality [13], our middleware architecture associates software modules directly with hardware modules. This allows the middleware to be tailored directly to the hardware platform's current configuration, resulting in a smaller code size.

Each software module is responsible for abstracting the hardware configurations of its corresponding hardware module.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

In particular, each software module is responsible for three tasks: 1) managing the appropriate hardware control and data registers, 2) performing the calculations associated with the module, and 3) managing the system's power. For example, the light-sensing software module is responsible for setting the control registers associated with controlling the MCU's analog to digital converter (ADC) and CPU. Since the sensing module is allowed to set the CPU's control registers, it can disable the CPU to save power while it is waiting for the ADC to settle on a value. Once the ADC has settled on a value, the sensing module can wake up the CPU and perform the necessary calculations.

All software modules are required to be stateless. This means that the libraries do not assume that the MCU's control or data registers are in a particular state. This is an important requirement since memory limitations prevent us from using an operating system to provide resource management and protection. The absence of an operating system would be concerning if the platform was required to run multiple programs simultaneously. However, our platform is designed for specialized embedded applications where resources such as memory are too limited to support an operating system.

Extending the middleware is relatively easy, since it is a collection of decoupled stateless modules. The stateless nature of these modules allows a developer to extend the platform without impacting the other libraries. If the software developer chooses to use these libraries in addition to controlling the registers, he or she is able to do so without impacting the middleware since each module is stateless. Regrading the design of the libraries, there are two seminal questions which we believe need to be addressed. The first question is how do we partition functionality between software and hardware and how does this affect the performance of the design? This problem of dividing the systems functionality between hardware and software is known as the "partitioning problem". Several researchers have studied this problem and have proposed ways to achieve the fastest or lowest cost solution [21] [40] [28] [14].

This implies that there are several partition options depending on the constraints of our design. So this leads us naturally to the second question. How will a software developer with limited hardware experience choose and configure these different hardware implementations? This is where the library abstraction proves to be extremely useful. One advantage of creating a direct mapping between the hardware modules and the software libraries is that new hardware/software partitions can be selected by including different libraries. This means that a software developer can select the appropriate partition by simply including the library that meets their performance and cost constraints, without having any knowledge of the underlying hardware.

III. Configuration Generation Process

In this section, we present an example that shows how a hardware configuration can be generated from a software definition. In particular, we go through a detailed example of how to implement an indoor temperature sensor with an LCD using the proposed code-first approach. The example code in figure 3 shows a program that was written in C using our modular middleware architecture. The program begins by including three software modules/libraries. The first is an LCD module which abstracts the display component. The second module is the temperature sensing module. And the third module is the helper module which includes helper functions. These helper functions include the $board\_init()$ function that abstracts the setup of the Watchdog Timer and the $convertADC()$ function that converts longs to formatted strings.

```
1  #include ``LCD.h''
2  #include ``tempSense.h''
3  #include ``helpers.h''
4
5  void main(void) {
6       unsigned long degrees;
7       char* reading;
8       board_init();
9       LCD_init();
10      while(1) {
11           LCD_gotoXY(0,2);
12           LCD_writeString(``AT:'');
13           degrees = tempSense();
14           LCD_writeChar(`(');
15           reading = convertADC(degrees,1);
16           LCD_writeString(reading);
17           LCD_writeChar(0x7f);
18           LCD_writeString(``C'');
19      }
20 }
```

Fig. 3. An example program written using the modular middleware that displays the temperature on the LCD.

Lines 10-19 represent the program's running loop. In the loop the program instructs the LCD screen to go to position (0,2) and write the string "AT". Once this step has completed, the program calls the tempSense module which sets the appropriate control registers, reads the appropriate data registers and returns the result, which is then converted to a string and displayed. The LCD software module supports unique characters with special codes, for example $0x7f$ represents the degree ^{o} character.

A. Introducing Mathematical Abstraction and Notation

There are currently solutions for partitioning software and hardware to create a performance optimized design using microcontrollers [14]. However, we were unable to find any research that addresses the problem of port conflicts that occur when implementing a collection of partitioned designs. A port conflict occurs when two partitions are included in the same design and require the same port. Figure 4 shows an intuitive example of a port conflict between two hardware partitions. If a port conflict occurs, it is not possible to implement the design given current hardware/software partitions. Engineers at Texas Instruments have developed a tool called pinMux that helps resolve conflicts in complex designs [5]. However, the port requirements must be specified before the software is written. There are also cases in which the pinMux tool is unable to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

find a solution and therefore port conflicts must be manually resolved by modifying the design to more efficiently use the ports. In this section, we present an approach for solving this port conflict problem by abstracting the problem as a constraint solving problem. The process is comprised of three steps:

1) abstracting the collection of hardware/software co-design libraries as a set, which represents the system's configured state;
2) representing the microcontroller port limitation as a collection of constraining set functions; and
3) applying the constraint set functions to the system set to automatically identify the software/hardware partition libraries that create port conflicts in the design.

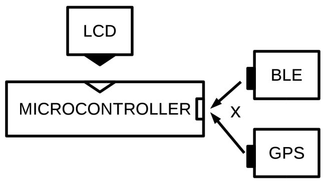


Fig. 4. Shows an example of a port conflict on the UART port of the microcontroller reference design. BLE represents a bluetooth module.

B. Abstracting Libraries

The first step in the process of identifying port conflicts is abstracting the hardware/software co-design libraries as a set representing the system's state. These state abstractions are best explained by an example. Throughout our explanation, we will use the code in figure 3 to show how each abstraction applies to the specific example.

We can abstract each library as a tuple t which is comprised of a library identifier λ and a set P which contains n properties such that P is a subset of the defined set P' which contains the set of all properties p. Table I shows an example of the set P'.

$$t = (\lambda, P = \{p_1, p_2..p_n\}) \tag{1}$$

We define t_λ as representing the library identifier of the tuple and t_p as representing the property set in the tuple. Each value p_i describes a particular property of the library. These properties are the requirements that the associated hardware partition needs to interface with the software library. Table I shows a collection of example properties. For example, p_1 may mean that the library is a UART based library, while property p_2 may mean that the component has an operating voltage of $3.3V$. Consider the "LCD.h" library from figure 3, it can be expressed in tuple form as: $(\lambda_3, \{p_3, p_4\})$. Table II shows a collection of example library identifiers and their associated properties.

Now that we have presented an abstraction for the software libraries and their associated properties, we can use this approach to model the software system S as a collection of tuples that are a subset of the defined set of tuples S'. Table II shows an example of the set S'.

property	description
p_1	UART module
p_2	Sensing ADC Module
p_3	3.3V power requirement
p_4	display module
p_5	on-board module
p_6	5v power requirement

TABLE I. AN EXAMPLE OF THE SET P'

library identifier	description	properties	include statement
λ_1	Bluetooth UART Library	$\{p_1, p_3\}$	"Bluetooth.h"
λ_2	IR Sensing	$\{p_2, p_3\}$	"IRSense.h"
λ_3	LCD display module	$\{p_3, p_4\}$	"LCD.h"
λ_4	Temperature Sensing module	$\{p_3, p_5\}$	"tempSense.h"
λ_5	GPS UART Library	$\{p_1, p_3\}$	"GPS.h"

TABLE II. AN EXAMPLE OF THE SET S'

$$S = \{(\lambda_1, P_1), (\lambda_2, P_2)...(\lambda_n, P_n)\} \tag{2}$$

Consider the following example code shown in figure 3. Now that we have defined the sets P' and S', we can determine the system's tuple representation by examining the library dependencies of the program. The program may contain other libraries that are not associated with hardware modules. By checking the libraries in the program against a known set of libraries it is possible to extract the relevant libraries from the program. Consider the program in figure 3 it has three library dependencies: "LCD.h", "tempSense.h" and "helpers.h". We construct the tuple set by checking for the library in the tuple set S'. If the library is found we add it to the set S. For example, the "tempSense.h" and "LCD.h" libraries are both in the set S' so we add them to the subset S. However, the "helper.c" library is not in the set S' so we do not add it to the subset S. This results in the following set:

$$S = \{(\lambda_3, \{p_3, p_4\}), (\lambda_4, \{p_3, p_5\})\} \tag{3}$$

C. Specifying the Constraints

Now that we have abstracted the hardware/software co-design libraries as a set representing the system's state, we need to represent the microcontroller's port constraints, so that they can be checked against the system's state. Recall that a port conflict occurs when there is a discrepancy between the system's state and the microcontroller's port constraints. For example, a program may include two libraries that have the UART property. This means that both libraries require the use of universal serial communication interface (USCI) module in the microcontroller. However, the msp430g2553 microcontroller does not have the required pins to support two UART devices, consequently even though the software definition of the program is correct, the hardware is unable to run it because the microcontroller does not have the required number of ports to support the design. In order to verify that a hardware configuration can be generated from the software model, we must show that the software model does not violate any of the hardware constraints. Formally, we can think of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

these constraints as a set of set functions

$$C = \{c_1(S), c_2(S)..c_n(S)\} \quad (4)$$

A constraint $c_i(S)$ is considered to be satisfied if it returns an empty set $c_i(S) = \{\}$. If a constraint is not satisfied, we say that it is violated. For a software model to be considered valid it must satisfy all of the constraints in the set C. Each hardware platform will have its own collection of constraints. For example, the proposed platform has a port constraint for UART modules p_1. The platform can only accommodate one UART module. We call this type of constraint a uniqueness constraint. Equation 5 shows an example of an uniqueness constraint for property p_1

$$c_1(S) = \{t | t \in S \wedge \exists t^{'} \in S : p_1 \in t_p \wedge p_1 \in t^{'}_p \wedge t \neq t^{'}\} \quad (5)$$

Another type of constraint is a universal property constraint. An example of a universal property constraint is a power constraint, which requires all modules to meet the 3.3V power requirement. Equation 6 shows an example of a universal property constraint for the property p_3.

$$c_2(S) = \{t | t \in S \wedge p_3 \notin t_p\} \quad (6)$$

We can define the violation set V as the set of the constraints that a system S violates.

$$V(S, C) = \{c | ((c \in C) \wedge c(S) \neq \{\})\} \quad (7)$$

We say a system S is consistent with a constraint set C if the violation set is empty. An empty violation set means that the system set does not violate any of the hardware constraints. This formalization is important because as the complexity of the system grows we expect that these abstractions will form the basis for explicitly specifying constraints, though we do not expect to supplant formal systems such as TLA [24]

Now that we have defined the concept of a constraint, let us consider it within the context of the hardware platform in figure 1. Because of the microcontroller's architecture and the design of the reference platform, the platform has three uniqueness constraints for properties p_1, p_2, p_4 and one universal property constraint for property p_3. Now let us consider how these constraints can be used to check the system set from our running example, shown in equation 3

To ensure that the set represents a valid software model, we must ensure that all constraints are satisfied. First, let us consider the uniqueness constraint shown in equation 5. The uniqueness constraint requires that a property p_i is only found in a single tuple in the set S. In the set S, the property p_4 occurs only once so the constraint is satisfied. Since properties p_1 and p_2 are not in the set, their uniqueness constraints are also satisfied. Now that we have shown that all the uniqueness constraints are satisfied, we need to show that the universal property constraint in equation 6 is satisfied. The universal property constraint requires that all tuples in S have the property p_i. This constraint is satisfied for property p_3 since all the tuples in S have the property p_3. Since all the constraints in C are satisfied, the software model is considered to be valid.

```
#include ``LCD.h''
#include ``tempSense.h''
#include ``Bluetooth.h''
#include ``GPS.h''
#include ``helpers.h''

void main(void) {
     unsigned long degrees;
     char* reading;
     char* gpsReading;
     board_init();
     LCD_init();
     while(1) {
          LCD_gotoXY(0,2);
          LCD_writeString(``AT:'');
          degrees = tempSense();
          LCD_writeChar(`(');
          reading = convertADC(degrees,1);
          LCD_writeString(reading);
          LCD_writeChar(0x7f);
          LCD_writeString(``C'');
          gpsReading = GPS_getValueString();
          bluetooth.send(reading);
          bluetooth.send(gpsreading);

     }
}
```

Fig. 5. Modification of indoor temperature sensor to include GPS and Bluetooth. For this example we assume the implementation of GPS module and it corresponding library.

It may be difficult to grasp the purpose of the constraints in a scenario where they are not violated. So let us consider a system where the constraints are violated. To see how the constraints could be violated, let us extend the temperature display example to include a GPS module and a bluetooth module. Figure 5 shows the code for the new design. By following the procedure outlined, we can define a new system set S_2 which presents this new GPS and bluetooth capable system. Equation 8 shows definition of the new set.

$$S_2 = \{(\lambda_3, \{p_3, p_4\}), (\lambda_4, \{p_3, p_5\}), (\lambda_1, \{p_1, p_3\}), (\lambda_5, \{p_1, p_3\})\} \quad (8)$$

Now we have a definition for the system set, we can once again apply the constraints. Notice that this time the uniqueness constraint is violated by both the GPS module and the bluetooth module, since the GPS module and bluetooth module both require the use of the UART ports and there are not enough pins on the microcontroller to accommodate. Notice also that the constraint does not simply return true or false but instead returns the violating tuples: $\{(\lambda_1, \{p_1, p_3\}), (\lambda_5, \{p_1, p_3\})\}$. These are then returned to the programmers as an error, notifying them of the libraries that have violated the limitation of the reference platform. Once these violation have been identified, the programmer may select new libraries that implement similar functionality but do not violate the port constraint of the microcontroller. For example, the programmer may choose a GPS implementation whose hardware/software partition uses an I^2C interface instead.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

library identifier	hardware id	mapped module
λ_1	α_1	Bluetooth UART module
λ_2	α_2	IR Sensing Module
λ_3	α_3	LCD Display module
λ_4	α_4	on-board temperature module
λ_5	α_5	Accelerometer Gyroscope module

TABLE III. An example of library mappings

hardware id	port id	port mapping
α_1	Υ_1	UART Port
α_2	Υ_2	Sensing ADC Port
α_3	Υ_3	Display Port
α_4	Υ_4	on-board no port
α_5	Υ_3	Sensing ADC Port

TABLE IV. An example of port mapping

D. Automating Microcontroller Selection

Currently our system is designed to be a proof of concept, but commercial alternatives would allow engineers to select from a wide variety of microcontrollers. This means that these constraints could be used to inform processor selection. If a particular microcontroller does not meet the constraints of the application, these constraints could be used to search for a microcontroller that might be capable of running the application. In this case, the problem becomes an optimization problem where the system is attempting to find the best possible microcontroller whose constraints meet the requirements of a given application. Since companies like Texas Instruments currently offer over 40 different microcontrollers [4], approaches that provide automatic microcontroller selection would be extremely useful.

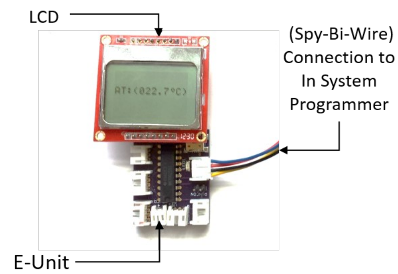
E. Generating the Hardware Configuration

Now that we have validated the software model, we can begin synthesizing the hardware configuration. The synthesis process is comprised of two steps. The first is to determine the list of hardware modules that are needed. The second step is to determine which ports each module plugs into. Our modular middleware architecture simplifies the first step by providing a mapping from the software libraries to the hardware modules. Table III shows the mapping of the libraries to the hardware modules. The design of the main board simplifies the second step by providing four unique expansion ports. Once we determine what hardware modules are required, we can look up the associated port in table IV.

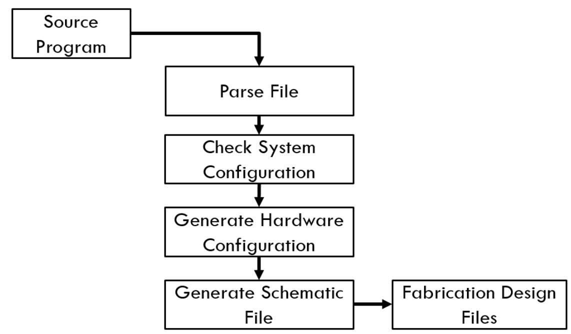
Now let us consider the example in figure 3, we determined that the software model for this code was $S = \{(\lambda_3, \{p_3, p_4\}), (\lambda_4, \{p_3, p_5\})\}$. Formally, we can think of the synthesis process as a set function that converts the software model set S to a hardware model set H. We define a hardware model set H as a collection of tuples,

$$H = \{(\alpha_1, \Upsilon_1), (\alpha_2, \Upsilon_4)...(\alpha_n, \Upsilon_m)\} \tag{9}$$

where α_i represents the hardware module id and Υ_i is the port identifier. We perform the first step of the synthesis process by identifying the corresponding hardware modules for libraries λ_3, λ_4 which are α_5, α_4 respectively. Now that we know what



(a) Temperature Sensor



(b) An overview of the process

Fig. 6. Figure (a) shows a temperature sensor with LCD display. Figure (b) shows an overview of the steps in automating the synthesis process.

hardware modules we need, we can look up the corresponding ports. This results in the tuple set $H = P\{(\alpha_3, \Upsilon_3), (\alpha_4, \Upsilon_4)\}$. From this configuration, the software developer knows to plug the LCD module into the display port and that the temperature sensor is already a part of the main board. The process of generating the hardware configuration from the software model can be easily automated since it is only a collection of look-ups. Figure 6(a) shows a picture of the resulting hardware configuration.

F. Automating the Process

In this subsection, we present the system architecture that implements the proposed formalization. Figure 6(b) shows an overview of the system's architecture. We implement an alpha version of the system by developing an eclipse plug-in that is compatible with Code Composure Studio [20].

The system is comprised of four stages: a parsing stage, a constraint checking stage, a configuration generation stage, and a schematic generation stage. A key insight for implementing the system is realizing that the super sets S, H and P can be represented as relational tables. By representing the sets as relational tables, it is possible to express and validate the system constraints as queries against these relations. This SQL-based architecture also allows the system to be updated as new libraries and constraints are added, since the database can be stored in a central location and queried remotely. However, before constraints can be validated using queries, the system set must be constructed. The system set is constructed by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

parsing all the files in the project directory and extracting the relevant include statements.

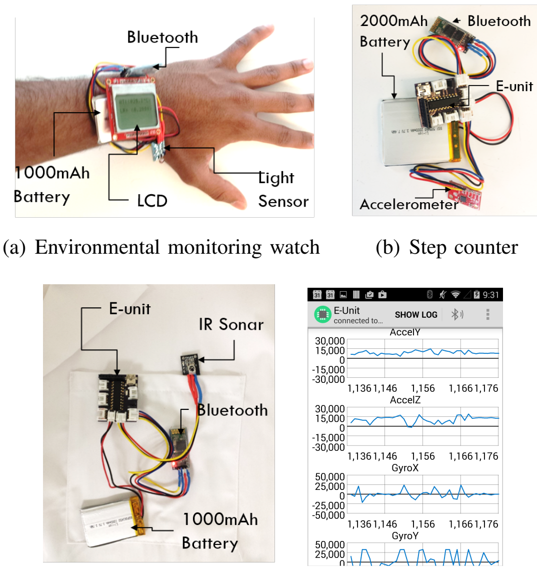
1) Parsing Step: During the parsing step the system reads all the files in the project folder and extracts all of the libraries that are included in the system. This is done by examining the include statements in the project files. Once all the libraries have been extracted, the libraries that are not associated with the middleware must be removed before the system definition can be generated. This list is filtered by querying the tables associated with the S relation. The libraries that exist within the P relation are kept and those that do not, are discarded. The remaining libraries represent the system set S. This system set is then represented as an arraylist of library ids, and is encapsulated as a member of a system node class.

2) Constraint Checking Step: Now that the system set has been determined, the next step is validating this system set. A valid system set is one that does not violate any of the constraints associated with the system. Since each constraint can be expressed as a single SQL statement, it can be further abstracted as a visitor class which operates on the system node. When a visitor operates on the system node it returns a violation set containing all of the libraries that violate the constraint that is associated with the visitor. This violation set is then added to a global violation set that is associated with the system node class. Once all the visitors have visited the system node, and if the global violation set is empty, the system set is considered valid.

3) Configuration Generation Step: Now that the system set has been validated, the next step is generating the hardware configuration. The hardware configuration specifies which hardware modules connect to a particular port. Recall that there is a one-to-one relationship between hardware modules and software libraries. This direct relationship allows us to query the S and H relations to determine the appropriate port and hardware module.

4) Schematic Generation Step: Once the programmer has finished testing the hardware prototype by configuring the platform, she may want to fabricate a custom hardware prototype. However, in order to do this she will need a custom schematic that represents her prototype. In this subsection we explain that it is possible to automatically generate a custom schematic using the proposed code-first approach. Schematic generation is possible because each hardware module maps directly to a schematic block and since each hardware module directly maps to a library, each library also maps to a schematic block.

Intuitively, this can be thought of as generating a schematic of the current hardware configuration while removing the unnecessary sections. A hardware schematic is normally represented using a .sch file. Fortunately, the .sch file is a xml based file. Since the schematic for each hardware module is known, each module can be represented as a xml block. Because of this, each block can be added as a child in the .sch xml files. Once the schematic block is added to the schematic, it needs to be appropriately wired. Given that the tree based hardware configuration already presents the appropriate wiring, the wiring for a particular schematic block can be determined by examining the port associated with the appropriate parent node. Since the connections from the ports to the modules are fixed, the appropriate wiring can simply be added to the schematic by looking at the wiring for the associated port.



(a) Environmental monitoring watch (b) Step counter

(c) Infrared system (d) Android application

Fig. 7. Figure (a) (b) (c) show the different possible configurations of the platform. Figure (d) shows a screenshot of the accompanying Android application

IV. Evaluation

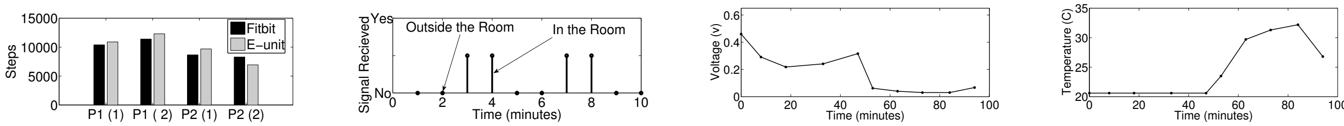
This section is divided into three subsections. In the first subsection, we evaluate the design by building a step counter that communicates with an Android application. In the second subsection, we evaluate the flexibility of the design by (1) prototyping a smartwatch that was designed to monitor the user's environmental exposure to temperature and light; and (2) prototyping an indoor tracking module which tracks the user's location using a collection of landmark infrared transmitters.

A. Step Counter

To test the system on a nontrivial example, we developed a prototype step counter and compared the results to the Fitbit. Figure 7(b) shows a picture of the prototype step detector. The prototype was programmed to use a windowed peak detection algorithm [7]. We evaluated the prototype by having two participants wear the step counter for two days between the hours of 9am to 9pm. The results of each participant are shown in figure 8(a).

To accompany the wearable platform we build an Android application that is compatible with the platform. Figure 7(d) shows a screenshot of the application. The application displays a real-time update of the readings that it receives from the platform. The readings are sent to the application from the module via Bluetooth. Once the Android application receives the values, it updates the appropriate section of the interface.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal



(a) Results of the step detection experiment (b) Readings from the IR sensor (c) Readings from the light sensor (d) Readings from the temperature sensor

Fig. 8. Figure (a) shows the results of the case study, in which two participants (Person1, Person2) wore both the prototype and a Fitbit for two days. Figure (b) shows the results of the localization experiment. When the signal is high it indicates that the prototype received the signal and when the signal is low it indicates that the prototype has not received the signal. Figures (c) and (d) show graphs of the light and temperature readings collected over a 1 hour period at 10 minute intervals. During the first 50 minutes the device was placed indoors and during the final 45 minutes the device was placed outside.

B. Environment Monitoring Smart watch

We prototyped a smartwatch designed to monitor the user's ambient temperature and light exposure. The watch was constructed using four components: the LCD display module, 3.7 V battery pack, the Bluetooth module and the mainboard. Once the components were placed on the mainboard, we developed software that captured the light and temperature readings and displayed them on the LCD screen. Figure 7(a) shows a prototype of the watch. Figures 8(c) and 8(d) shows the results of the tests.

C. Infrared Indoor Localization Device

The third device that we prototyped was an indoor infrared localization device. This wearable device localizes an individual by using unique infrared signatures from landmark infrared devices. Each landmark device is placed in a separate room and produces a unique infrared signal. As the user moves from room to room an infrared sensor located on the wearable device picks up the unique infrared signature of the landmark device. Since this infrared signature is unique to each room, this information can be used to localize the user.

We used our platform to quickly prototype this device using the mainboard and three modules: the infrared sensor, the Bluetooth module and the 3.7V battery. Figure 7(c) shows a picture of the final prototype. This prototype is designed to fit in the user's shirt pocket, with the infrared sensor sitting slightly above the rim of the pocket so that it can pick up the infrared signatures from the landmark devices. The wearable device communicated with a smart phone using the Bluetooth module. However, if users would like to explore a lower power option they may elect to remove the Bluetooth module and replace the large 3.7V battery with the smaller 3.3V coin battery. Though the Bluetooth module is compatible with the 3.3V voltage battery, the smaller 3.3V coin does not have the capacity to sustain the Bluetooth module for long periods. So instead of using the Bluetooth module to transmit readings to a smart phone, the software developer can record the signature by writing the signature values to the chip's internal flash memory. These values can be retrieved later by connecting the platform to a PC and reading the chip's internal flash memory. In our evaluation, we used an infrared LED to present the landmark devices. We tested the device by walking into the room with the landmark device for two minutes and then walking out and waiting two minutes. The device was polled at one minute intervals. Figure 8(b) shows the results of the experiment.

System	Library ID
Watch	"Bluetooth.h" , "LCD.h", "tempSense.h", "lightSense.h"
IR system	"Bluetooth.h", "irSense.h"
Step Counter	"AccelerometerRead.h" ,"Bluetooth.h"

TABLE V. SYSTEM AND ASSOCIATED LIBRARIES

Recall that constraints are fixed and are associated with the processor and do not change based on the application.

Constraint Type	Properties
Universal Property Constraint	p_1, p_2, p_4
Uniqueness Constraint	p_3

TABLE VI. SYSTEM AND ASSOCIATED LIBRARIES

D. System's Limitations

One of the requirements of this approach is that the hardware/software partitions are sufficiently isolated. If the components are not sufficiently isolated they can affect the performance of the microcontroller and the other components. We tested this isolation requirement by connecting a motor controller circuit that was not sufficiently isolated to the E-unit. We also connected a bluetooth module that was sufficiently isolated. We noticed failures in the bluetooth module when both components where operated at the same time.

Another limitation of the system is the fixed power requirement. Since the ports on the reference design only provide 3.3V, it is not possible to connect a module that has a 5V requirement. Other researchers have proposed a custom port design that has two power supplies: a 3.3V supply and 5V [38]. Components that connect to a reference design simply select the appropriate power pin. Adopting this alternate design may help address these issues, but may also increase the complexity of the schematic generation process.

E. Considerations for a Commercial Solution

1) Interrupt Vectors: Interrupt vectors may prove problematic since the libraries are stateless and therefore cannot rely on the state of the microcontroller's interrupt vectors. This can be resolved by using wait loops (spin locks) in place of interrupts. These spin locks limit the performance of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

system where real-time performance is required. However, this can be mitigated by using a real-time operating system with appropriately partitioned hardware/software libraries [6].

2) Operating Systems and Languages : In an ideal case we would like to support a variety of operating systems and languages. Supporting languages like python and java would make the approach more accessible to a wider variety of developers. However, the memory limitations of the microcontroller used in our prototype make it difficult to run an operating system and java virtual machine, without which, it is impossible to run java byte code.

3) Repository of Reference Designs: Facilitating a commercial implementation of the schematic generation process would require a repository of partitioned hardware/software co-design libraries that adhere to a specific format. Developing this repository would require significant engineering effort as both software libraries and hardware schematics would have to be developed for each additional module, before it could be leveraged by the synthesis process.

V. Related Work

Research into rapid prototyping strategies can be divided into two major categories: fixed platform approaches and hardware/software co-design approaches. Fixed platforms use predefined designs, while hardware/software code design approaches use co-synthesis strategies to generate efficient hardware and software partitions.

The hardware/software co-design approaches that are the most similar to our design can be grouped into two subcategories: 1) Interface-Based Designs and 2) Platform-Based Designs. One of the earliest papers on Interface-Based Design by J. Rowson et al. proposed a methodology for separating a component's behavior from how it communicates with other components in the system. Separating components in this way makes it easier to formally verify the component's behavior [33]. Since then, several researchers have explored this interface based design paradigm [30] [29] [10]. Though our platform uses an interface based approach that is similar to previously proposed approaches, our approach extends the interface based paradigm beyond the verification of a single component to the synthesis of an entire system.

The second hardware/software co-design approach is a Platform-Based Design approach. A Platform-Based approach encourages the reuse of pre-designed components through the use of automatic mapping tools [34] [22]. These automatic mapping tools use a layered approach to isolate and map an abstracted top layer description to a more detailed lower layer implementation. Consider the example of a field programmable gate array which uses a compiler to provide isolation and automatic mapping from the top layer VHDL abstraction to the lower layer implementation of logic blocks. In our approach, the libraries provide the abstraction for hardware/software partitions and the automatic mapping to lower level hardware implementation is done by the automatic configuration generation process.

In addition to many hardware/software co-design approaches, several fixed platform based approaches have also been proposed. Several companies and researchers have developed platforms that allow researchers and engineers to quickly prototype and test their new ideas. One of the earliest prototyping platforms was the phidgets platform, which was developed in 2001 [15]. Afterwards, in 2003, Plessl et al. advocated for the inclusion of field programmable gate arrays (FPGAs) in sensor nodes [31]. They presented a sensor hardware architecture which coupled an fpga with a CPU. By including the fpga and allowing the CPU to configure it, they were able to dynamically configure the chips. Our approach does not use a reconfigurable chip. Instead, we focus on extending the capabilities of the platform by adding and removing external modules.

The earliest occurrence of an extensible platform that we found in the literature was the MetaCricket. [26]. The MetaCricket consisted of a main board which connected to other devices and sensors. The board consisted of a main master controller and a supporting slave controller. The external sensors and expansion boards connected to the master controller while the slave controller controlled the board's internal components. Following the release of MetaCricket, researchers at Stanford University introduced the GoGoBoard in 2004 [35]. The GoGoboard was a low cost programmable control and sensing board. The GoGoBoard was designed to be used as a learning resource in developing countries. With this goal in mind, the researchers focused on ensuring that the GoGoBoard could be assembled in developing countries. This decision influenced the board's design and the components that were selected. Unlike the GoGoBoard and the MetaCricket, our platform consists of both hardware and software components which make the process of prototyping easier.

In 2005, researchers at UC Berkeley proposed the Telos platform along with the TinyOS operating system [25]. The release of the reference platform and operating system sparked innovation in sensor network research. Following the release of the Telos platform, researchers at the UC Berkeley released an updated platform called the TelosB. And in 2009, researchers at UC Berkeley [23] extended the capacity of the TelosB motes by creating two extension boards. The first extension board comprised of a triaxial accelerometer and a biaxial gyroscope. The second extension board provided electrocardiogram (ECG), and electrical impedance pneumography (EIP) functionality to the TelosB platform. These expansion boards demonstrated the flexibility of the TelosB motes. However, as new computing form factors emerge, such as body networks and wearable devices, researchers will need a platform that allows them to prototype devices for these new applications. Unlike the Telos and TelosB motes, our proposed platform can be extended using off-the-self components instead of custom extension boards. More recently, in 2014, researchers at the University of Florida have developed a reconfigurable RFID sensing tag [27]. The platform provides three pins that can be used to connect sensors to the platform. The platform also has an RFID antenna and Cortex M3 microcontroller.

Large companies have also attempted to develop reconfigurable platforms. For example, in 2008 Shimmer began developing their wearable computing development kit [8]. The Shimmer kit was a flexible health sensing kit which consisted

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2016.2537148, IEEE Internet of Things Journal

of a collection of prebuilt expansion modules. Following the release of the Shimmer development kit, researchers at Microsoft proposed the Gadgeteer platform in 2011. The Gadgeteer platform is an extensible platform that allows researchers and industry professionals to quickly prototype hardware devices [38]. The Gadgeteer's main board is designed to use a collection of prebuilt modules which can be plugged into the main board. Unlike the Gadgeteer and Shimmer platforms, our proposed platform does not require custom modules. It works with off-the-shelf devices.

VI. Conclusion

In this paper, we presented a code-first approach to designing embedded systems. The proposed approach allows software developers to create hardware prototypes by first writing the code that the prototype is required to run. Once the code has been written, it can be analyzed and an appropriate hardware configuration can be generated. By using library abstractions to create a one-to-one mapping between software and hardware modules, it is possible to determine the hardware dependencies by examining the libraries included in the program. However, there are several challenges associated with this approach. For example, limitations in the microcontroller's architecture could make a given hardware configuration infeasible, even though it is running a valid program. For example, a port conflict may occur when two libraries, whose associated hardware modules use the same port, are both included in a program. In this case it is not possible to generate a viable hardware configuration because there are not enough ports to accommodate all of the modules. To address this issue, we presented an approach for automatically identifying port conflicts. We evaluated our design by building three prototypes: 1) a step counter, 2) an environmental monitoring smartwatch, and 3) an infrared indoor location system.

References

[1] fitbit flex product page. https://www.fitbit.com/flex. Accessed: 2014-07-8.

[2] Pebble Smartwatch pebble steal. https://getpebble.com/steel. Accessed: 2014-07-8.

[3] Samsung Galaxy gear. http://www.samsung.com/us/mobile/wearable-tech/SM-V7000ZKAXAR. Accessed: 2014-07-8.

[4] Texas Instruments development kits and software for low-power mcus. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/tools_software.page. Accessed: 2015-10-14.

[5] Texas Instruments pinmuxtool. http://www.ti.com/tool/PINMUXTOOL. Accessed: 2015-10-14.

[6] Richard Barry. *Using the FreeRTOS real time kernel: a practical guide*. Real Time Engineers, 2010.

[7] Agata Brajdic and Robert Harle. Walk detection and step counting on unconstrained smartphones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and ubiquitous computing*, pages 225–234. ACM, 2013.

[8] Adrian Burns, Barry R Greene, Michael J McGrath, Terrance J O'Shea, Benjamin Kuris, Steven M Ayer, Florin Stroiescu, and Victor Cionca. Shimmer–a wireless sensor platform for noninvasive biomedical research. *Sensors Journal, IEEE*, 10(9):1527–1534, 2010.

[9] Pablo Castro, Sergey Melnik, and Atul Adya. Ado. net entity framework: raising the level of abstraction in data programming. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1070–1072. ACM, 2007.

[10] Pai Chou, Ross Ortega, Ken Hines, Kurt Patridge, and Gaetano Borriello. ipchinook: An integrated ip-based design framework for distributed embedded systems. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 44–49. ACM, 1999.

[11] G De Michell and Rajesh K Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[12] Stephen Edwards, Luciano Lavagno, Edward A Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Readings in hardware/software co-design*, 86, 2001.

[13] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[14] Rolf Ernst, Jörg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *Readings in hardware/software co-design*, pages 18–29, 2002.

[15] Saul Greenberg and Chester Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM, 2001.

[16] Rajesh K Gupta and Glovanni De Micheli. Hardware-software cosynthesis for digital systems. *Design & Test of Computers, IEEE*, 10(3):29–41, 1993.

[17] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, and Cory Sharp. Flexible hardware abstraction of the ti msp430 microcontroller in tinyos. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 277–278. ACM, 2004.

[18] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, and David Culler. Flexible hardware abstraction for wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 145–157. IEEE, 2005.

[19] Jason L Hill and David E Culler. Mica: A wireless platform for deeply embedded networks. *Micro, IEEE*, 22(6):12–24, 2002.

[20] Texas Instruments. Code composer studio users guide. *Texas Instruments Literature Number SPRU328B*, 2000.

[21] Asawaree Kalavade and Edward A Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48. IEEE Computer Society Press, 1994.

[22] Kurt Keutzer, Jan M Rabaey, A Sangiovanni-Vincentelli, et al. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[23] Philip Kuryloski, Annarita Giani, Roberta Giannantonio, Katherine Gilani, Raffaele Gravina, V-P Seppa, Edmund Seto, Victor Shia, Curtis Wang, Posu Yan, et al. Dexternet: An open platform for heterogeneous body sensor networks and its applications. In *Wearable and Implantable Body Sensor Networks, 2009. BSN 2009. Sixth International Workshop on*, pages 92–97. IEEE, 2009.

[24] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[25] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[26] Fred Martin, Bakhtiar Mikhak, and Brian Silverman. Metacricket: A designer's kit for making computational devices. *IBM Systems Journal*, 39(3.4):795–815, 2000.

[27] Mohammad S. Islam Muhammad S. Khan and Hai Deng. Design of a reconfigurable rfid sensing tag as a generic sensing platform toward the future internet of things. *IEEE Internet of Things Journal*, 1(4), 2014.

[28] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

[29] Roberto Passerone, Luca De Alfaro, Thomas A Henzinger, and Alberto L Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 132–139. ACM, 2002.

[30] Roberto Passerone, James A Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the 35th annual Design Automation Conference*, pages 8–13. ACM, 1998.

[31] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, Lothar Thiele, and Gerhard Tröster. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, 7(5):299–308, 2003.

[32] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369. IEEE, 2005.

[33] James A Rowson and Alberto Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th annual Design Automation Conference*, pages 178–183. ACM, 1997.

[34] Alberto Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, 2002.

[35] Arnan Sipitakiat, Paulo Blikstein, and David P Cavallo. Gogo board: augmenting programmable bricks for economically challenged audiences. In *Proceedings of the 6th international conference on Learning sciences*, pages 481–488. International Society of the Learning Sciences, 2004.

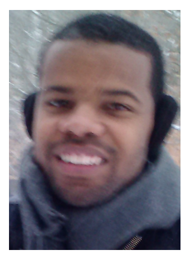
[36] Mani Bhushan Srivastava and Robert W Brodersen. Rapid-prototyping of hardware and software in a unified framework. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 152–155. IEEE, 1991.

[37] Karl Van Rompaey, I Bolsens, Hugo De Man, and D Verkest. Cowarea design environment for heterogenous hardware/software systems. In *Proceedings of the conference on European design automation*, pages 252–257. IEEE Computer Society Press, 1996.

[38] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. . net gadgeteer: a platform for custom devices. In *Pervasive Computing*, pages 216–233. Springer, 2012.

[39] Mark Weiser and John Seely Brown. The coming age of calm technology. In *Beyond calculation*, pages 75–85. Springer, 1997.

[40] Wayne H Wolf. Hardware-software co-design of embedded systems [and prolog]. *Proceedings of the IEEE*, 82(7):967–989, 1994.



Daniel Graham is a fifth year student pursuing a Doctorate Degree in Computer Science at William and Mary. He is working with Dr. Gang Zhou, and his research interests include intelligent embedded systems and networks. He received his M.Eng. in Systems Engineering from the University of Virginia in 2011, and his B.S. from the University of Virginia in 2010.

Gang Zhou Dr. Gang Zhou is an Associate Professor in the Computer Science Department at the College of William and Mary. He received his Ph.D. degree from the University of Virginia in 2007 under Professor John A. Stankovic. He has published over 70 academic papers in the areas of ubiquitous computing, mobile computing, sensor networks, and wireless networks. The total citations of his papers are more than 4500 according to Google Scholar, among which five of them have been transferred into patents and the MobiSys04 paper has been cited 800 times. Thirteen of his papers have each attracted more than 100 citations since 2004. He is currently serving in the Journal Editorial Board of IEEE Internet of Things as well as Elsevier Computer Networks. He received an award for his outstanding service to the IEEE Instrumentation and Measurement Society in 2008. He also won the Best Paper Award of IEEE ICNP 2010. He received NSF CAREER Award in 2013. He is a Senior Member of ACM and also a Senior Member of IEEE.