Ensuring Safety of Model-based Generated Code for Pervasive Health Monitoring Systems

by

Sunit Verma

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2013 by the
Graduate Supervisory Committee:

Sandeep Gupta, Chair
Cihan Tepedelenlioglu
Martin Reisslein

ARIZONA STATE UNIVERSITY

May 2013

ABSTRACT

Wireless technologies for health monitoring systems have seen considerable interest in recent years owing to it's potential to achieve vision of pervasive healthcare, that is healthcare to anyone, anywhere and anytime. Development of wearable wireless medical devices which have the capability to sense, compute, and send physiological information to a mobile gateway, forming a Body Sensor Network (BSN) is considered as a step towards achieving the vision of pervasive health monitoring systems (PHMS). PHMS consisting of wearable body sensors encourages unsupervised long-term monitoring, reducing frequent visit to hospital and nursing cost. Therefore, it is of utmost importance that operation of PHMS must be reliable, safe and have longer lifetime. A model-based automatic code generation provides a state-of-art code generation of sensor and smart phone code from high-level specification of a PHMS. Code generator intakes meta-model of PHMS specification, uses codebase containing code templates and algorithms, and generates platform specific code. *Health-Dev*, a framework for model-based development of PHMS, uses code generation to implement PHMS in sensor and smart phone. As a part of this thesis, model-based automatic code generation was evaluated and experimentally validated. The generated code was found to be safe in terms of ensuring no race condition, array, or pointer related errors in the generated code and more optimized as compared to hand-written BSN benchmark code in terms of lesser unreachable code.

ACKNOWLEDGEMENTS

I own sincere thankfulness to my advisor Dr. Sandeep K.S. Gupta for giving me an opportunity to work at the Impact lab (http://impact.asu.edu/) and his support and encouragement in guiding me in my thesis work. I thank my committee members Dr. Martin Reisslein and Dr. Cihan Tepedelenlioglu for their valuable advice and suggestions in improving my thesis. I am indebted to all the members of Impact lab especially Dr. Georgios Varsamopoulos, Dr. Ayan Bannerjee, Priyanka Bagade, Joseph Milazzo, Yu Xie, Joshua Ferguson, and Madhurima P ore for their constant help and support. I thank National Science Foundation (through grants CNS-0831544 and IIS-1116385) for funding the research. I thank my parents, brother, my roommates and friends for their love and support.

To my parents for all their love and sacrifices

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

Chapter 1
INTRODUCTION

Increase in the cost of hospital-based care and worldwide deployment of wireless networks and infrastructures has encouraged the development of Pervasive Health Monitoring Systems (PHMS). Figure 1.1 shows PHMS consist of resource constraint body-wearable sensors or medical devices that gathers physiological information from the body and communicates with the base station [1]. Base station controls the sensors and as well as uploads the data to a cloud server via wireless link which can be accessed by physician for further diagnosis and reference. PHMS makes continuous monitoring system of human physiology easy and deployable. Continuous monitoring helps in avoiding confusion between fatal and non-fatal symptoms such as, heart-attack and heart-burn can can similar symptom of chest pain. Hence, a user can detect the source of symptom only if his/her body is being continuously monitored.

In order to have pervasive healthcare, anyone should be able to make a PHMS at anytime and anywhere with a guarantee of safe operation of PHMS software. Since PHMSes usually operate in an unsupervised scenario, software operating such PHMSes plays an

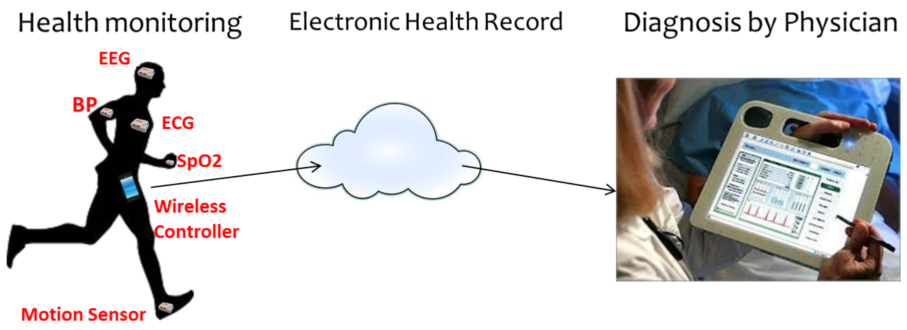


Figure 1.1: Pervasive Health Monitoring System (PHMS).

important role in monitoring and controlling of human physiology. In a typical PHMS, software is concurrent, distributed and runs on a memory constrained sensor having no hardware-based memory protection. Such software are often evaluated so that developers, regulators, physicians and patient can be confident in PHMS's safety. A PHMS is considered as safe if it's operation does not cause any harm to the body of user. However, software-related errors in PHMS have been attributed to the failure of life saving medical devices. Following errors [2] may occur due to various reasons:

- Rule-based error : These errors occur when software development approach is correct but poor implementation results in logic failure in the software. As medical devices industry is growing at faster pace, demand to perform rapid implementation, preferably manually, increases the risk of involving potential errors in the software of medical devices. There has been several recent cases of failures of life saving medical devices such as drug delivery systems, as reported by Food and Drug Administration (FDA) Maude database [3].

- Memory failure : Memory failure occur due to access of restricted area in memory. As most of the embedded devices, such as medical sensors, are memory constrained it becomes difficult to carefully manage the memory usage.

- Knowledge-based error : These errors occur due to a poor software development approach. It can also be caused due to inadequate knowledge to implement algorithms which process and adapt to critical changes in human physiological data.

- Abnormal use : Abnormal use of medical software without any user guide may cause user interface of a medical device controller to crash resulting to interruption in monitoring and may also cause loss of physiological data.

Among these four sources of software-related errors, rule-based error and memory failure have been identified as commonly occurring errors in PHMS software [4]. In this thesis I have tried to ensure rule-based error and memory failure don't occur in a PHMS software. A safe PHMS software must ensure no memory and logic failure occurs during PHMS operation.

Most of sensors used in PHMS are run on TinyOS [5], a low-power operating system which is used to target platforms of wireless sensor network applications. A typical PHMS application can have thousand lines of code that are concurrent and distributed. They are often

- run on memory-constrained sensors with no hardware protection to memory
- run for a long duration of time
- have to cope with unpredicted situations such as radio out-of-range, sensor failure etc.

These constraints makes it difficult to avoid race condition and array & pointer-related errors in TinyOS [6]. Hence a safe PHMS software must ensure no race, out-of-bound array access and null-pointer dereferencing occurs.

This thesis focuses on an automated software development of PHMS to reduce the potential errors in manual implementation. In this regards, a model-based automatic code generator for sensor and smart phone was developed. The code generator takes PHMS model as an input from a high-level specification and applies code generation to platform-specific template code to generate device-specific code. Using this framework, the generated code was evaluated and validated based on the requirements set forth in specification phase. *My contribution was to evaluate the code generated against BSNBench code [7]* *and validate against set forth requirement in specification phase. The code was further applied to a static code analyzer to observe the extent of optimization achieved in terms of less unreachable code and safety by ensuring no arrays and pointers related errors in the generated code.*

The rest of the thesis is organized as follows - Chapter 2 presents discussion of approaches to develop PHMS software. It also describes the TinyOS execution model and various code generation techniques; Chapter 3 describes the proposed model-based code generator for PHMS; Chapter 4 discusses *Health-Dev*, a model-based development of PHMS which uses code generator to generate sensor and smart phone code; Chapter 5 evaluates and validates the generated code; Chapter 6 concludes the thesis with a discussion on future works.

Chapter 2
BACKGROUND

This chapter discuss manual and automated implementation approaches to develop PHMS software. Most of the sensors used in PHMS are run on TinyOS [8, 9]; hence, for generation sensor code, code generator must be aware of TinyOS execution models. Unsafe TinyOS programming scenarios that leads to memory and logic failure are discussed so that code generation must avoid them to ensure safety in generated code. Finally, various source code generation techniques are also discussed.

2.1 PHMS SOFTWARE DEVELOPMENT APPROACH

To make a continuous monitoring system where a sensor senses ECG (Electrocardiography) signal, calculate heart-rate and send the processed data to smart phone via Bluetooth as shown in Figure 2.1 , there are two approaches to implement the system:

- Manual
- Automated

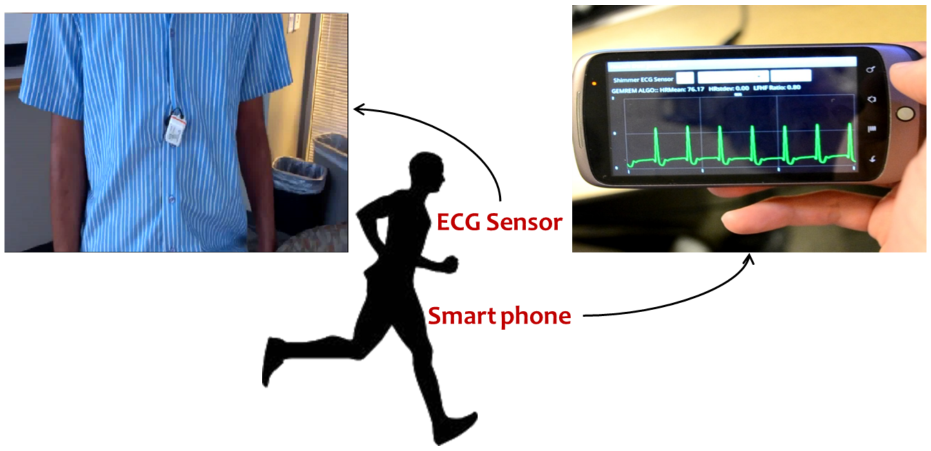


Figure 2.1: ECG Monitoring System.

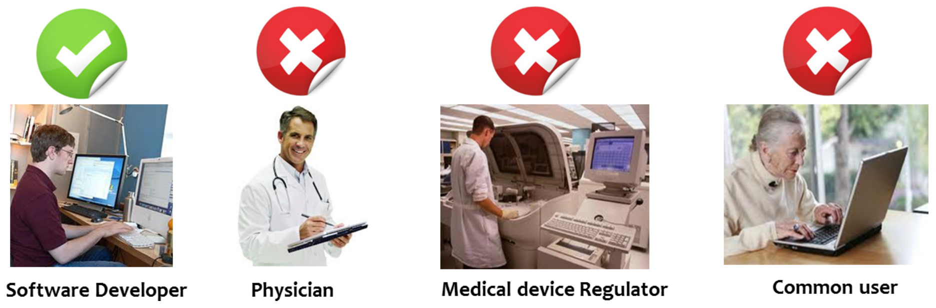


Figure 2.2: Only software developer can perform manual implementation.

Manual implementation

Manual implementation requires user to have domain-specific programing knowledge such as C-based languages, Java, etc. Only a software developer can implement the system as shown in Figure 2.2, but the implementation time and quality also depends on the skills and experiences of software developer. A physician can have in-depth knowledge on diagnosis of a patient but may not have knowledge to implement a customized PHMS. Medical device regulator can have extensive knowledge of PHMS design requirements but may not have the knowledge to customize the PHMS accordingly. A common user can be a health concerned person who just wishes to monitor health quickly with no hassle, but may not have PHMS implementation knowledge.

In 2011, it was found that software failures were responsible for 24% of medical device failures [10] where logic failure and corruption in RAM (Random Access Memory) were commonly found errors. One of the main causes of such failure were attributed to manual implementation of PHMS [11].

Automated implementation

Automated software development can be one of the solutions to avoid memory and logic failures in PHMS implementation. The automated implementation must generate sensor and smart phone code from a high-level specification. It must be easy for anyone (de-

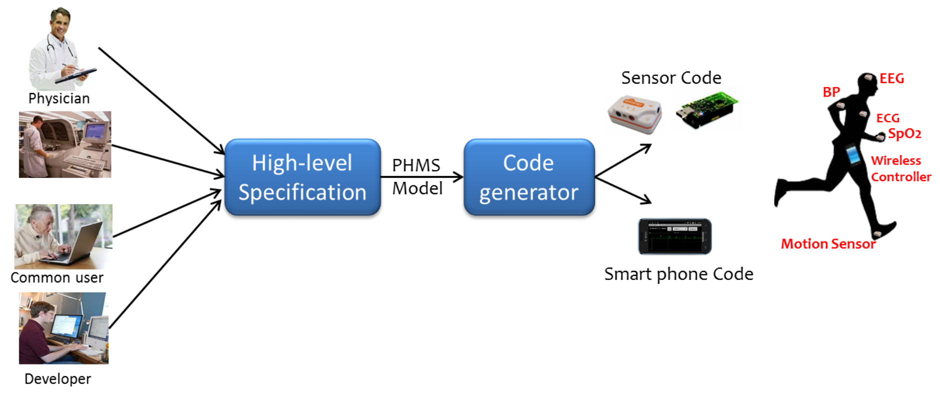


Figure 2.3: Automated software development of PHMS software.

veloper, physician, medical device regulator and common user) to specify the high-level specification as shown in Figure 2.3. The code generator must generated safety-assured code to ensure safer operation of PHMS.

This approach can reduce development cost and time as well as provide rapid prototyping from high-level specification of system models. As such, PHMS implementation can be performed by physicians, medical device manufacturer and common user, who may not necessarily understand each others design perspective. Hence, it is important to reduce the complex design process by using high-level specification. Automatic software development has been adopted in real time systems and non-real time systems. PHMS is a real time system where each request is required to be processed in a given time constraint. Many researchers have proposed automatic code generation methodologies for real-time embedded systems [12, 13, 14, 15, 16, 17, 18, 19, 20, 21] and amongst them model-based approach has received considerable focus [12, 14, 18, 19, 20, 21]. A model-based approach deals with the exploitation of abstract representation of a complex system and also helps in compliance checking and testing of models before actual implementation. Model-based code generation exploits the models of a complex systems e.g. PHMS to generate executable code. It also allows easier maintainability and readability in generated code.

Automatic code generation of sensor and smart phone is a complex process due to diversities in hardware and software components of sensor and smart phone as shown in Figure 2.4. Although all sensors are bound to perform sensing, computation and communication with smart phone, they have different types of sensing modules, micro-controllers, communication radios and above all, they may have different operating systems (OS). Some of the most preferred OS for sensor platforms are TinyOS which is based on event driven paradigm and Linux-based e.g. iMote. Hence, programing abstraction for sensors are diverse. In the case of smart phone, software development is typically done using Software Development Kit (SDK) which is available as a library to high-level languages such as C or Java. SDK contains hardware abstraction; uniform across different platform for a given OS. However, SDK supports different programming languages which may vary in syntax and semantics. Hence, for each sensor platform and OS specific smart phone, code generator must have template files in a comprehensive code base to understand the programing paradigm and hardware abstraction in order to generate the code.

Figure 2.4 describes an overview of an automatic code generator for PHMS. A high-level specification must allow user to specify sensor and base station platform. User must be able to specify sensing, communication and algorithm properties for the respective platforms. A model parser must convert the specification into sensor and base station models which is input to the code generator creating required sensor and base station code.

2.2 TinyOS EXECUTION MODELS

TinyOS applications are written in nesC, a dialect of the C language, optimized for memory constrained sensor networks. TinyOS has two-level scheduling: event (interrupt) and task. It is called an event-driven operating system in which events performs small hardware-based processing, and long running tasks are interrupted.

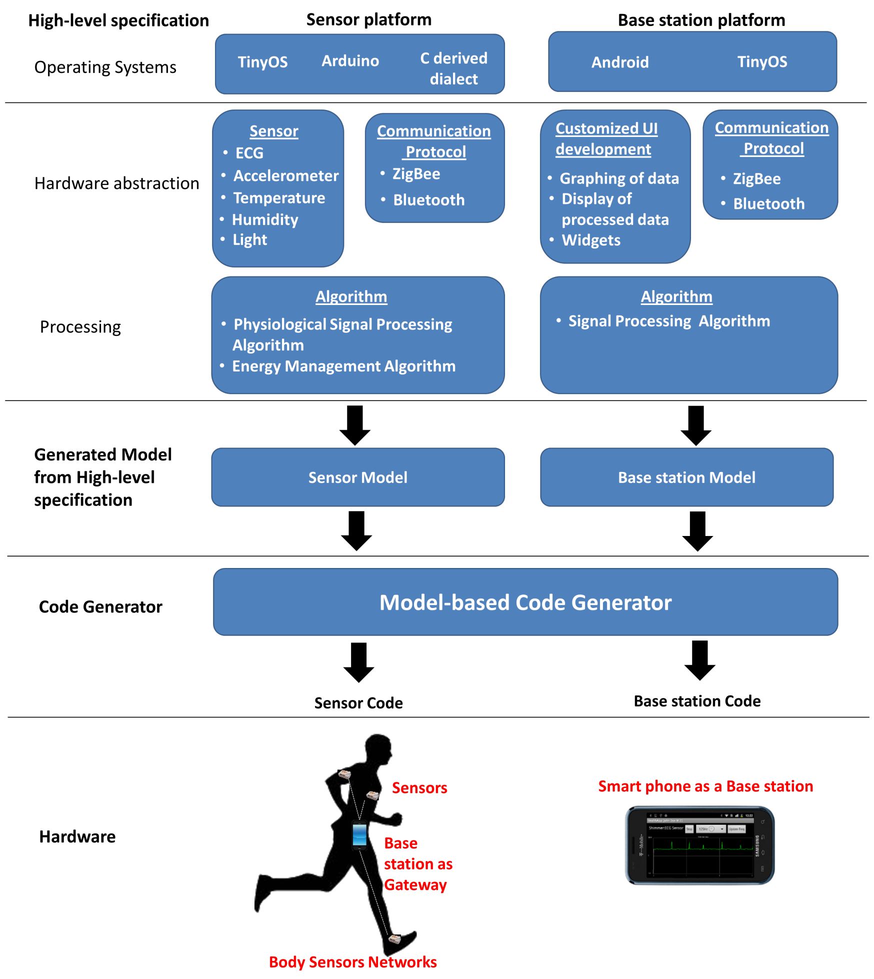


Figure 2.4: Automatic Code Generator extracts software and hardware requirements from the models to generate the code for PHMS.

Event

Event represents hardware-based interrupts such as clock and radio and does a small amount of data processing. Clock-based events (timer interrupt) occurs periodically while radio interrupts occurs when radio of a sensor receives data from other sensor or a base station. The Figure 2.5 shows an architecture of event implementation in TinyOS. TimerC component sends a call command to ClockC component. The call command is a non-blocking request to lower-level component. An event will be invoked by ClockC and the event handler in

TimerC will process the event in first-in, first-out order. For example, in the following code snippet, ReadECG.read is called to read the sensed data. An event ReadECG.readDone responds to the call stating if the data was successfully read or not.

```
if (call ReadECG.read() != SUCCESS) {
report_problem();
}
event void ReadECG.readDone(error_t result, uint16_t data) {
if (result != SUCCESS) {
           report_problem();
      }else{ . . . . . . . . . .}
}
```

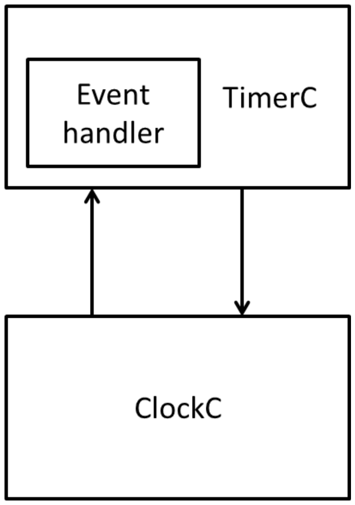


Figure 2.5: Architecture of event in TinyOS.

Task

Task handles larger data computation which take a longer amount of time and are run in background. Tasks are a form of deferred procedure call (DPC), which enables a program to perform a computation or operation until a later time. They are synchronous and always run to completion. However, they can be interrupted by any event as they have low priority. For example, in the following code snippet, when event TimerECG is fired, a task is posted to perform data computation which runs in the background. The Table 2.1 summarizes the difference between Task and Event.

```
task void ecg_data_compute() {
 \*perform computation*\
}
event void TimerECG.fired() {
  if (ECG_reading == NREADINGS_ECG) {
      // Post a task to perform processing of data
      post ecg_data_compute();
}
}
```

Table 2.1: TinyOS execution model.

Task	**Event**
Deferred background processing	Handles hardware interrupts
Less priority	High priority
Synchronous code	Asynchronous code
Non-preemptive	Preempt Tasks and Events
Called by Event or Call command	Called by Call command

TinyOS execution flow of ECG sensing and processing

This section gives an overview of the flow of sensed data in a sensor before it is sent over Bluetooth to smart phone. The Figure 2.6 gives a flow diagram of TinyOS execution in ECG sensor. The Boot event calls Bluetooth radio to start and also starts a periodic timer.

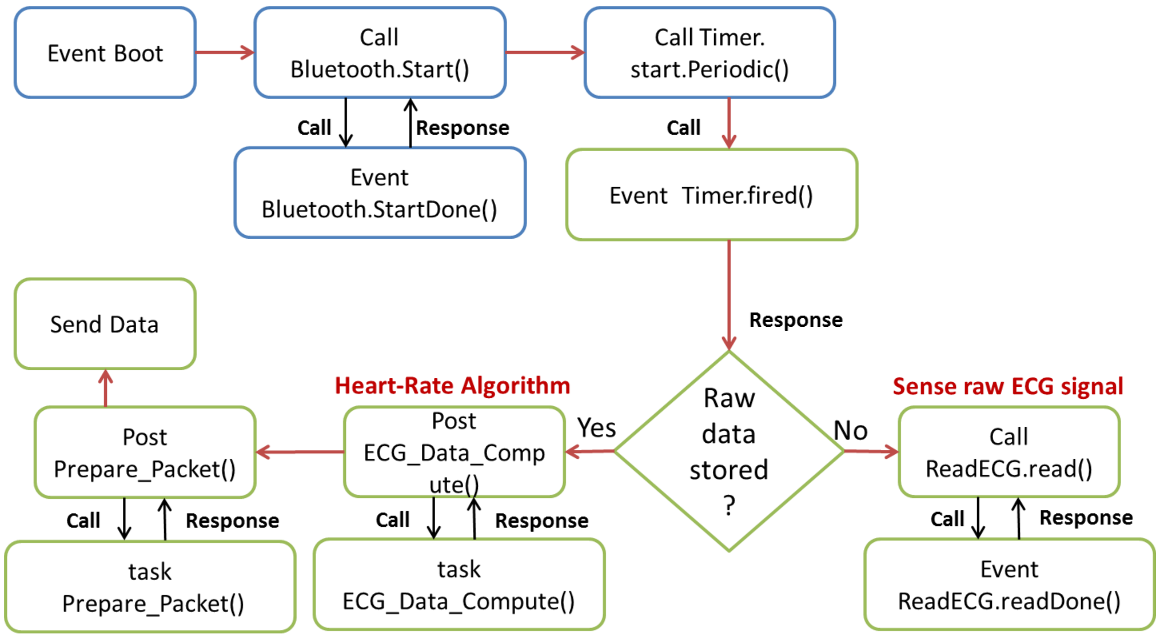


Figure 2.6: Execution flow of ECG sensing and processing in a sensor.

Whenever Timer is called, an event is fired as response to the call. The fired event checks if raw data is available in a buffer. If raw data is not available, ReadECG is called to read and store the data. A task is posted to calculated heart-rate from raw data and another task is posted to prepare the packet of data to send it via Bluetooth to smart phone.

2.3 Safety hazards in TinyOS programming

TinyOS programming becomes unsafe mainly due to the following errors which affects the memory of micro-controller.

- Null-pointer dereference : For a given TinyOS sensor code, if a structure is defined with a uninitialized pointer to the structure and if there is an attempt to write data using that pointer, then memory corruption can occur. This occurs due to the fact that the data will be written on to the output register P1OUT (MSP430 micro-controller) of peripheral of port P1, potentially modifying the operation of those modules controlled through P1OUT bits.

```
struct sensor_data {
   nx_uint16_t  size;
   nx_uint16_t  data[10];
} *sensor_data;          // pointer not initialized
// write into sensor_data->data[9]
sensor_data->data[9] =  rawData[1];  // memory corruption
```

- Out-of-bound array access : Attempt to access out-of-bound array indexes can corrupt the stack as well as adjacent stack memory. In the following code snippet, there is an attempt to copy data from rawData to 10^{th} index of data array but the index value supported by data array is from 0 to 9.

```
sensor_data->data[10] =  rawData[1];
```

- Race conditions : Whenever task (synchronous) and event (asynchronous) or two events simultaneously perform read/write operation to a data structure, race condition occurs. Operation on the shared data must be mutually exclusive otherwise the possibility of corrupting the shared data increases. For example, in the following code snippet, sensor_data is shared between task and event which can potentially incur a race condition when accessing sensor_data. This is because a task is not run immediately and thus it is unclear when sensor_data is going to be access first. The probability of race becomes highly likely when two tasks share sensor_data.

```
event void TimerECG.fired() {
    sensor_data->data[9]= rawData[1];
    post ecg_data_compute();
    post ecg_prepare_packet();
}
task void ecg_data_compute() {
    sensor_data = outputData;
}
task void ecg_prepare_packet() {
    sendData = sensor_data;
    .........
}
```

In order to generate safety-assured code for PHMS, an automatic code generator must ensure above errors don't occur in code generation.

2.4 CODE GENERATION TECHNIQUES

With the use of model-based development, new applications can be created with less effort and time than traditional approaches. The principles of model-based development are successfully applied in avionic and auto manufacturing industries [22]. Lately, software engineering industry has suffered due to unmanageable complexities in the process of product development. This led model-based development to become a major focus in software engineering. Generation of source code is becoming an integral part of software engineering especially in the context of model-based development approach. Researchers have proposed various approaches and tools for source code generation which is subsequently compiled and run [23].

TEMPLATES + FILTERING

In this approach, the filter is applied to a higher-level specification to extract relevant information as specification subset (Figure 2.7). The generator applies the template code to the specification subset resulting in target code. The filtering mechanism must be powerful and specification must be clearly defined to allow code generation. Since this approach is tightly coupled with specification syntax, industry standard tool such as Extensible Markup Language (XML) and Extensible Stylesheet Language Transformations (XSLT) can easily act as a high specification language.

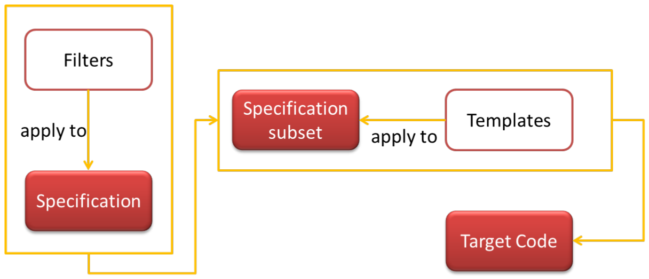


Figure 2.7: Template with filtering.

TEMPLATES + META-MODEL

This approach focuses on code generation from a meta-model which is a simplified description of a model (Figure 2.8). The generator parses the high-level specification which adheres to meta-model and creates an instance of the meta-model. The code templates are applied on it to generate the target code. One of the advantages of this approach is parsing and instantiation of the models are completely separate from code templates and hence it makes easier to change the format of the specification (model) as long as generated meta-models are in compliance with code generator. Such approach can be used for domain-specific applications.

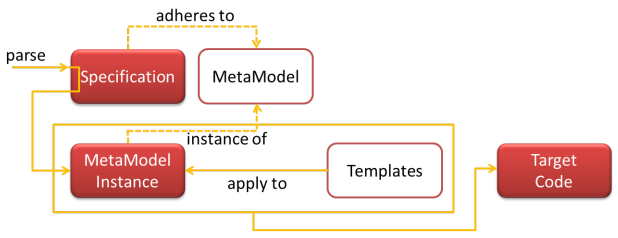


Figure 2.8: Template with meta-model.

FRAME PROCESSING

This approach focuses on code generation with the usage of parameterizable templates called frames (Figure 2.9). A frame can be viewed as typed function that facilitates code generation. The code generation is controlled by one frame called as specification frame which instantiates, parametrizes and composes other frames. The frame processing could be achieved from two methods; script-based frame processor and adaption approach. In script-based approach, script instantiates and parametrize frames. In adaption approach, the frame processor injects code into specific location in other frames.

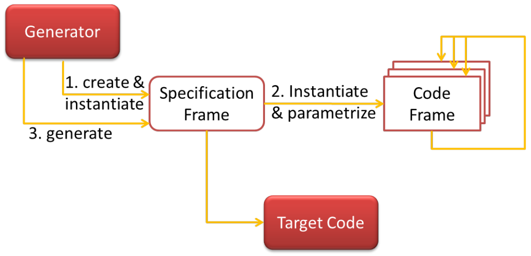


Figure 2.9: Frame processing.

API-BASED GENERATION

To handle generation of small pieces of code for a well-defined task, Application Programmer Interface (API), which are defined in terms of abstraction of the code can be used (Figure 2.10). The developer will manually write the program with the help of APIs to create or modify the code. The APIs are formulated in terms of a standard grammar. The generated code will also be an instance of that grammar. Such approach is best suited where generation of small code are required.

INLINE CODE GENERATION

This approach uses preprocessing technique to generate the required code. The source code with variant configuration is preprocessed based on configuration specification (Figure 2.11). Source code remains in an iteration until all the required preprocessing is achieved.

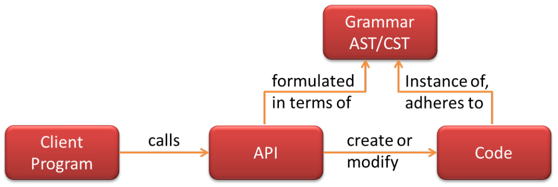


Figure 2.10: API-based generation.

When all variants are resolved the code is generated. With a facility to change variables, conditions, and type expressions accordingly, one can generate customized and optimized code from a single template file where several versions of source code are embedded. Such approach is used when developing a library which is required to run on multiple platforms.

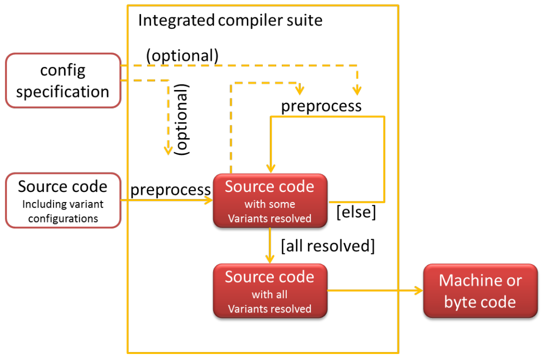

Integrated compiler suite
config specification
(optional)
(optional)
preprocess
Source code Including variant configurations
preprocess
Source code with some Variants resolved
[else]
[all resolved]
Source code with all Variants resolved
Machine or byte code

Figure 2.11: Inline code generation.

CODE ATTRIBUTES

In this approach, the source code is annotated with attributes. The generator parses the source code and creates required changes to source code based on attributes (Figure 2.12). The attributes are usually special comments specifying what functionality or expression should replace the comment. One can annotate in any fashion as long the parser understands the annotation.

CODE WEAVING

The basis of this approach is derived from API-based code generation technique (Figure 2.13). There are different meta-artifacts which are used for code composition. Each meta-

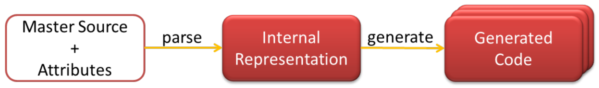


Figure 2.12: Code attribute.

artifact has different functionality and must be considered how each meta-artifact influences others. A code-weaver must take care of above constraints before weaving the pieces of code together.

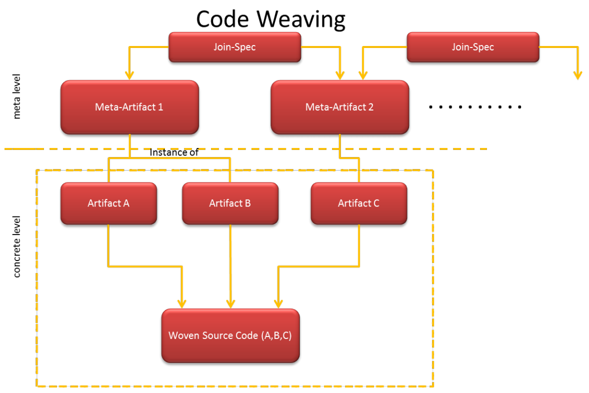


Figure 2.13: Code weaving

Chapter 3
PROPOSED CODE GENERATOR

To design the code generator for PHMS, there are various requirement aspects that needs to be undertaken. The code generator must be flexible and manageable from developers point of view and accepts any high-level specification tool given the meta-model derived from it maintains compliance with code generator. Following are the characteristics of the code generator for PHMS.

3.1 MODEL-BASED

The model-based development allows modeling of a software system from a higher abstraction level. The model can be directly transformed to a programming language with the help of automatic code generation. User will be able to specify PHMS specification without worrying about low-level details of implementation. It can be more cost-effective as it reduces development time and provides more opportunities to test each of the functionalities to ensure they are less error prone. Hence compliance checking and testing of model can be easily performed. Based on the different approaches of code generation discussed, only TEMPLATES+META-MODEL provide model-based code generation.

3.2 MANAGEABILITY

One of the important aspects of code generator in the context of its development is that code generation framework must incorporate flexibility and extensibility. Such aspect facilitate the developer an ability to change the functionality without making major changes to the code generation framework. Similarly, extension of hardware and software abstraction can be implemented in modular fashion. Hence, modularity provides better management in internal components of code generator. As discussed in various approaches to generate code, only TEMPLATE+META-MODEL and FRAME PROCESSING approach stresses on modularization.

3.3 Algorithm specification

The code generator for PHMS must support physiological signal processing algorithms, the declaration of algorithm and provision of its execution sequence; play a critical role in ensuring real-time operation without missing a deadline. Such requirement calls for modularity in specifying algorithm which can be accomplished by introducing algorithms as frames. As mentioned in FRAME PROCESSING approach, frames are functions which are instantiated and parametrized before its usage. Such approach will allow the ability to inject algorithm frames into a specific location and bring modularity to template code. Only FRAME PROCESSING provides such modular approach.

3.4 Annotation in template code

The code template should provide the information to parser about what and where the frames should be injected. Template code should have annotations or special comments to make parser of a generator understand the code accordingly. The benefit of this pattern is one can annotate code with as many information as long as parser of a code generator understands it. Only CODE ATTRIBUTES provides such approach.

3.5 Flexibility and extensibility in template code

From re-usability point of view, for a given template code, the template must be able to provide several versions of code. The template code should be flexible which can allow embedment of several versions of templates ensuring it runs on diverse platform. Extensibility aspect should be available to include new sensor or communication support to the same platform. With the help of pre-processing directives, such properties can be achieved. Only INLINE CODE GENERATION provide such approach.

3.6 Model-based code generator for PHMS

Currently, there exist no approach for code generation of PHMS that addresses all the characteristics discussed, model-based code generator for PHMS was proposed (Figure 3.1) which inherits all the necessary properties from different code generation approaches to make it compatible for PHMSes.

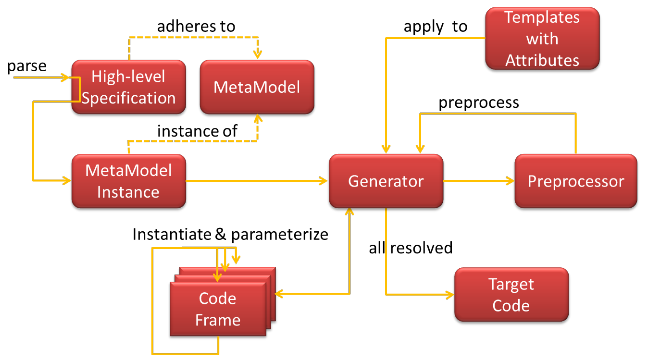
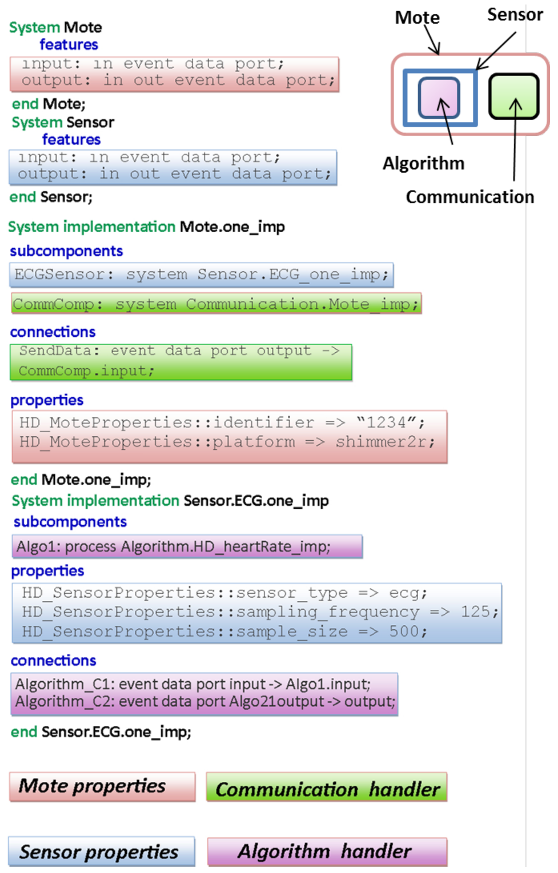


Figure 3.1: Model-based code generator for PHMS.

The high-level specification adhering to the meta-model is parsed to create an instance of the meta-model and is fed to generator as input. Based upon the given meta-model, the generator pulls a copy of a platform-specific template code with attributes. It applies preprocessing technique to template file, defining and expanding the required expressions. Appropriate code frames are instantiated and parametrized before injection into a specific locations in template file. The code generator is divided into modules as follows:

High-level specification

The high-level specification allows user to specify requirements to implement PHMS without worrying about low-level details of implementation. The specification must adhere with the meta-model format which is the actual input to the generator module. The specification must allow user to specify sensor and smart phone specification and translate it to respective models. Architecture Analysis and Design Language (AADL) is one such language used for model-based engineering in embedded system engineering that allows the modeling of an embedded systems. It allows user to define constructs for sensors and smart phone to create a system architecture. The Figure 3.2 shows that a sensor construct is defined with input and output ports. The instance of that sensor consists of an identifier, algorithm and communication components. It also allows to specify routing of the data through algorithms.

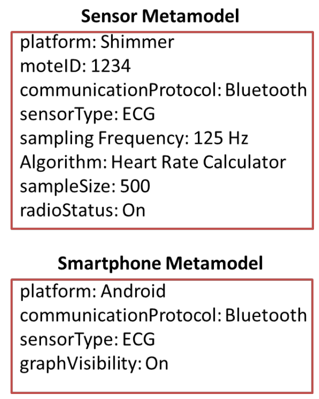


```
System Mote
    features
 input: in event data port;
 output: in out event data port;
end Mote;
System Sensor
    features
 input: in event data port;
 output: in out event data port;
end Sensor;
System implementation Mote.one_imp
subcomponents
 ECGSensor: system Sensor.ECG_one_imp;
 CommComp: system Communication.Mote_imp;
connections
 SendData: event data port output ->
 CommComp.input;
properties
 HD_MoteProperties::identifier => "1234";
 HD_MoteProperties::platform => shimmer2r;
end Mote.one_imp;
System implementation Sensor.ECG.one_imp
subcomponents
 Algo1: process Algorithm.HD_heartRate_imp;
properties
 HD_SensorProperties::sensor_type => ecg;
 HD_SensorProperties::sampling_frequency => 125;
 HD_SensorProperties::sample_size => 500;
connections
 Algorithm_C1: event data port input -> Algo1.input;
 Algorithm_C2: event data port Algo21output -> output;
end Sensor.ECG.one_imp;
```

Figure 3.2: Sensor model in AADL.

Meta-Model

Meta-model contains bare minimum but necessary information of the model. It is an abstraction of a model which is itself an abstraction of real-life system. Meta-modeling provides methods to analyze, create constructs, frames, and rules to model a given set of problems. The high-level specification is written under such regulation, which allows parser to create a meta-model from given model. The parser extracts out all the relevant information from the model, which is further used in code generation. Figure 3.3 shows meta-model containing information which allows for code generation.



Sensor Metamodel

platform: Shimmer
moteID: 1234
communicationProtocol: Bluetooth
sensorType: ECG
sampling Frequency: 125 Hz
Algorithm: Heart Rate Calculator
sampleSize: 500
radioStatus: On

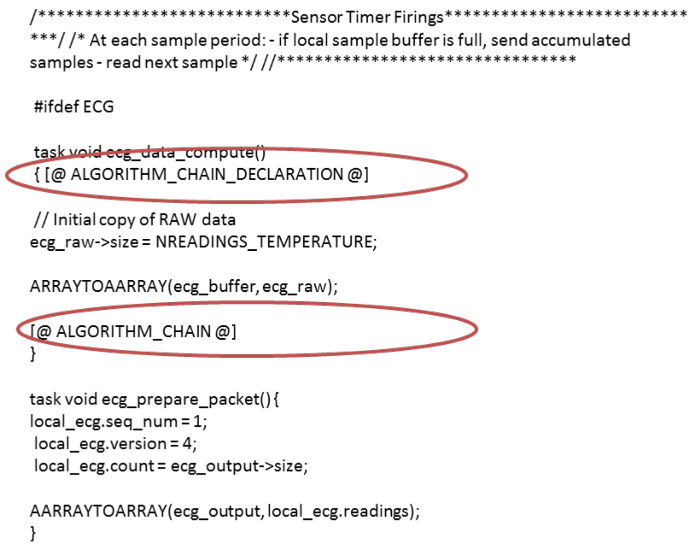
Smartphone Metamodel

platform: Android
communicationProtocol: Bluetooth
sensorType: ECG
graphVisibility: On

Figure 3.3: Sensor and smart phone metal-model are input to generator.

Template code with attribute

Template code is platform-specific code which acts as a base to generate required code. It contains reference entities or attributes which helps the parser to inject required frame (piece of a code) and brings modularity in generation. This also makes template code lightweight and generic by storing the functionalities in separate modules, which can be easily modified or replaced. Given the information obtained from meta-model, the generator calls an appropriate template code and replaces its attributes with respective frames. The frame can be an algorithm or a function. In the Figure 3.4, the encircled code snippets are attributes of a TinyOS template code that shows where to declare an algorithm and call for sequencing.


```
/**************************Sensor Timer Firings**************************
***/ /* At each sample period: - if local sample buffer is full, send accumulated
samples - read next sample */ //********************************

 #ifdef ECG

 task void ecg_data_compute()
 { [@ ALGORITHM_CHAIN_DECLARATION @]

 // Initial copy of RAW data
ecg_raw->size = NREADINGS_TEMPERATURE;

ARRAYTOAARRAY(ecg_buffer, ecg_raw);

[@ ALGORITHM_CHAIN @]
}

task void ecg_prepare_packet() {
local_ecg.seq_num = 1;
 local_ecg.version = 4;
 local_ecg.count = ecg_output->size;

AARRAYTOARRAY(ecg_output, local_ecg.readings);
}
```

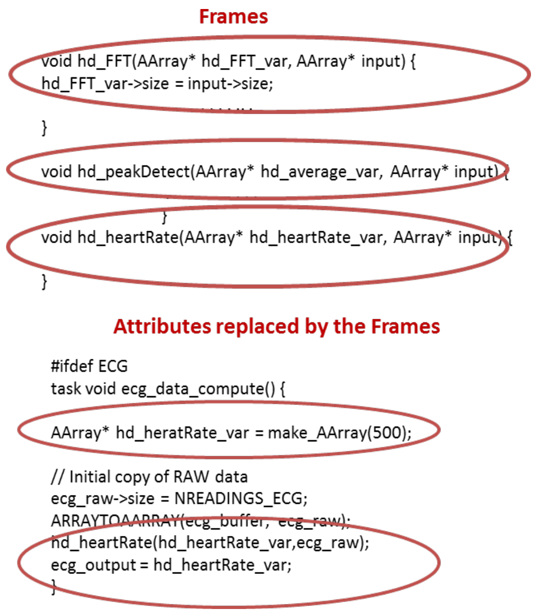
Figure 3.4: Code snippet of TinyOS template with attributes.

Code frame

Code frames are generic implementation of a function which could be used by generator after they are parametrized. They are used for specifying and sequencing of algorithms. The generator injects the frame into the specific location, making the generation more modular. Figure 3.5 shows a repository of algorithm frames namely, Peak detection, Fast Fourier Transform (FFT) and Mean. These frames are declared and sequenced in a copy of a template code by generator. Figure 3.4 shows the template with attributes indicating where the algorithm should be declared; generator injects the frames in particular location after instantiation.

Preprocessor

The Preprocessor provides the ability to include user-defined header files, macro expansion, and conditional compilation into the template code which are evaluated during compile time. This makes the generated code customized for specific applications. Based on the information obtained from meta-model and template code, the generator calls the pre-



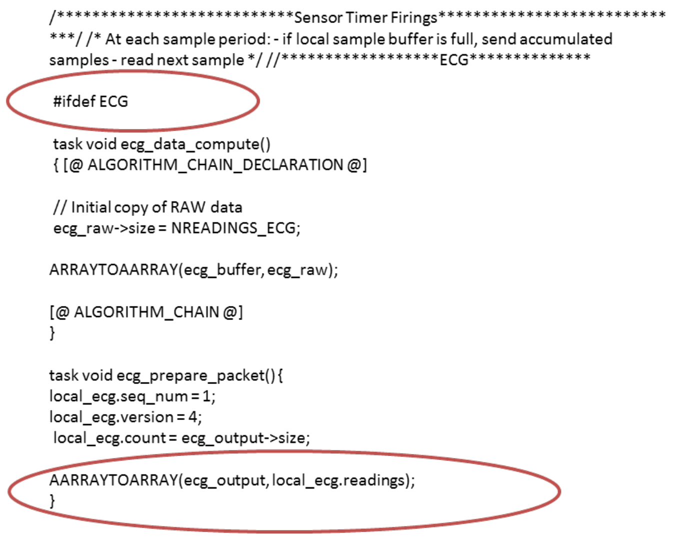
```
void hd_FFT(AArray* hd_FFT_var, AArray* input) {
hd_FFT_var->size = input->size;
......
}
void hd_peakDetect(AArray* hd_average_var, AArray* input) {
}
void hd_heartRate(AArray* hd_heartRate_var, AArray* input) {

}
```

```
#ifdef ECG
task void ecg_data_compute() {
AArray* hd_heratRate_var = make_AArray(500);
// Initial copy of RAW data
ecg_raw->size = NREADINGS_ECG;
ARRAYTOAARRAY(ecg_buffer, ecg_raw);
hd_heartRate(hd_heartRate_var,ecg_raw);
ecg_output = hd_heartRate_var;
}
```

Figure 3.5: Declaration and sequencing of algorithm frame into TinyOS template code.

processor to enable required sensor types, communication protocol, energy management, and macro expansions. In Figure 3.6, the encircled code snippet are preprocessor directives which show a sensor type is enabled and macro expressions that copies array of raw data to an output array.



```
/***************************Sensor Timer Firings**************************
***/ /* At each sample period: - if local sample buffer is full, send accumulated
samples - read next sample */ //*****************ECG**************

 #ifdef ECG

 task void ecg_data_compute()
 { [@ ALGORITHM_CHAIN_DECLARATION @]

 // Initial copy of RAW data
 ecg_raw->size = NREADINGS_ECG;

ARRAYTOAARRAY(ecg_buffer, ecg_raw);

[@ ALGORITHM_CHAIN @]
}

task void ecg_prepare_packet() {
local_ecg.seq_num = 1;
local_ecg.version = 4;
 local_ecg.count = ecg_output->size;

AARRAYTOARRAY(ecg_output, local_ecg.readings);
}
```

Figure 3.6: Preprocessors and macros in TinyOS template code.

Generator

Generator outputs the required code from a given meta-model. Generator calls template code for sensor and smart phone, uses preprocessing and code frame modules to generate the code.

3.7 Safety-assurance in PHMS software from code generator

As the code generator is aware of the of the execution model; safe operation of PHMS software is allowed, by ensuring none of the discussed error occurs.

- No null-pointer dereferencing : Whenever a structure is defined, pointers to structure are initialized before they are used in the following code. When a piece of code requires to use the structure, a pointer to the structure is allocated with the required memory before it is used so that no null-pointer errors occur.

```
typedef struct Algorithm_Array {
int32_t* data;
int16_t size;
} AArray;

AArray* ecg_raw =NULL;
AArray* ecg_output =NULL;

inline AArray* make_AArray(int size) {
AArray *ptr = (AArray*) malloc(sizeof(AArray));
ptr->data = (int32_t*) malloc(sizeof(int32_t) * size);
ptr->size = (int16_t) malloc(sizeof(int16_t));
return ptr;
}
ecg_raw = make_AArray(SAMPLE_SIZE_OF_RAWECG);
ecg_output = make_AArray(OUTPUT_SIZE_OF_DATA);
AArray* heartRate_var = make_AArray(500);
   . . . . . . .  . .
 /* Perform heart-rate data computation*/
 . . . . . . .  . .
ecg_output = heartRate_var;
```

For example, in the above code snippet, before using ecg_output, it is initialized and allocated with required memory.

- No out-of-bound array access : To ensure no array-related errors occur, code generator defines generic macros which facilitates data transfer between two data structures.

```
int32_t ecg_buffer[SAMPLE_SIZE_OF_RAWECG];
// Initial copy of RAW data
ecg_raw->size = SAMPLE_SIZE_OF_RAWECG;
ARRAYTOAARRAY(ecg_buffer, ecg_raw);
//This copies a standard array into the data array in AArray struct.
#define ARRAYTOAARRAY(a, b) {
for (arrayIndex = 0;  arrayIndex < b->size;  arrayIndex++) {
         b->data[arrayIndex] = a[arrayIndex];
       }
}
```

As shown in the above code snippet, in order to transfer the data from ecg_buffer (array) to ecg_raw (pointer to structure) a macro ARRAYTOAARRAY will ensure safe data transfer as the number of data transfered is limited to the size of the array, ensuring no out-of-bound array access occur.

- No race conditions : Race conditions are avoided by ensuring no data structure is shared between task and event as well as between two events as shown below. Shared data structures are avoided by posting task in event code which handles data processing related to that data structure.

```
event void TimerECG.fired() {
   post ecg_data_compute();
}
task void temperature_data_compute() {
   hd_heartRate( hd_heartRate_var, ecg_raw);
   ecg_output = hd_heartRate_var;
}
```

Chapter 4
HEALTH-DEV

Health-Dev allows model-based development of PHMS by automatically generating sensor and smart phone code from a high-level specification of requirement verified PHMS design [12]. It uses *Ayushman* [1] as a test bed. *Health-Dev* allows user to design PHMS using an intuitive Graphical User Interface (GUI). The GUI allows user to easily specify PHMS specification without having the knowledge of system architecture and low-level implementation details. The GUI translates the specification into AADL architectural model. The architectural model is parsed to generate meta-models which are used in the code generation module for actual implementation. The code generator uses meta-models with codebase to generate code for sensor platform and smart phone. Figure 4.1 gives an overview of *Health-Dev* architecture. It consists of following modules:

4.1 SPECIFICATION MODULE

The GUI module of Health-Dev acts as the specification module. The GUI works by loading an AAXL file which is a XML representation of architectural AADL model. AAXL is represented as Document Object Model (DOM) Tree in the GUI module. The user is presented with multiple screens where input acts as the specification. GUI translates the inputs from user into AAXL properties and makes the proper modifications by writing it to AAXL file which later is used to generate AADL model of the system. As shown in Figure 4.2, the GUI allows user to add a mote, base station, and network component. The components can be edited or deleted based upon user's preference. Each component has specifiable properties which allows user to customize accordingly.

The main screen of GUI contains three tabs namely, Mote, Base station and Network

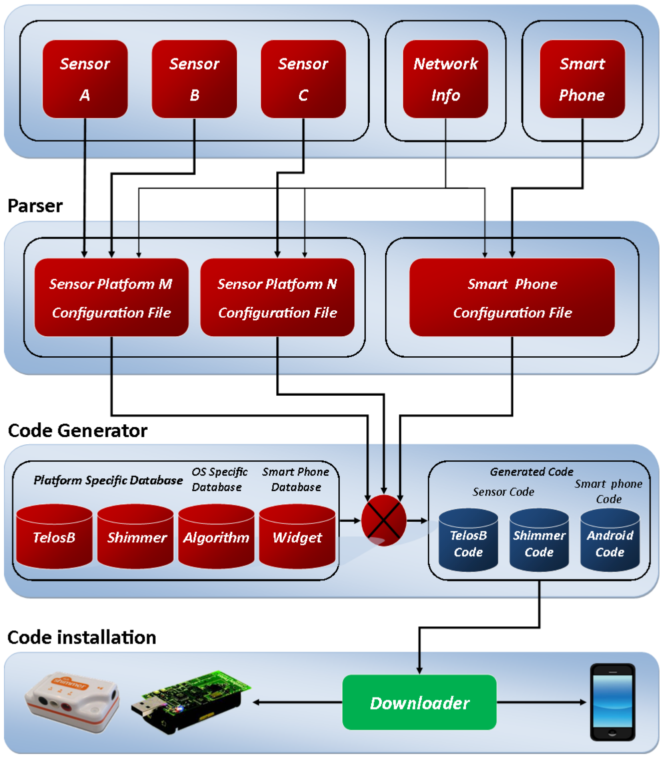
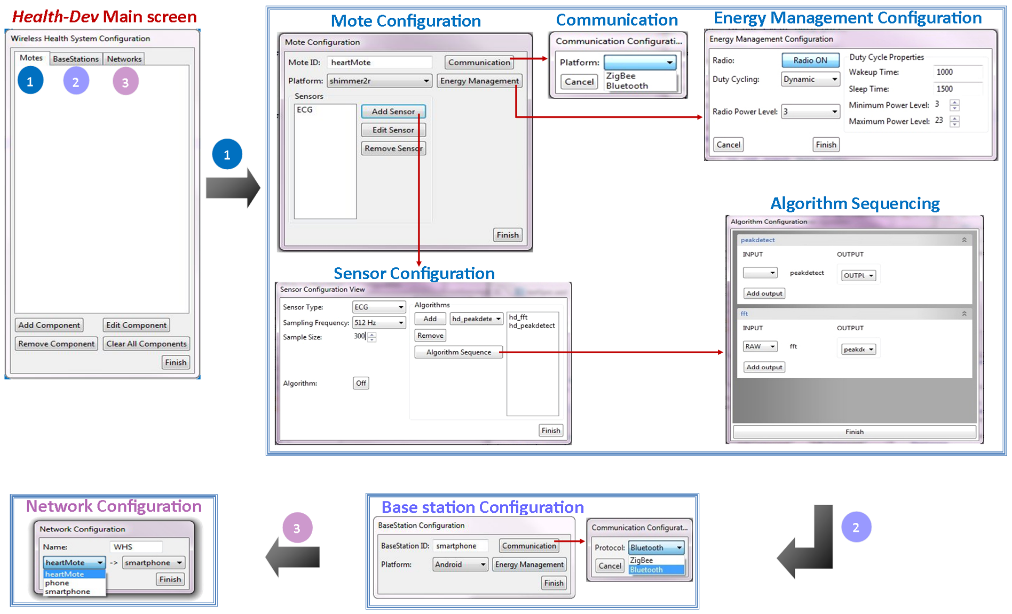


Figure 4.1: Health-Dev Architecture Overview.

which gives the overview of PHMS by keeping it simple. Each tab acts as a main component of its category and can be modified by using Add, Edit, Remove and Clear all components options. GUI allows user to implement PHMS by clicking on a particular tab and adding a component to it e.g By adding a component to Mote tab allows user to configure Sensor, Algorithm, Communication and Energy management subcomponents. Similarly, user can configure Base station and Network as shown in the Figure 4.2. The GUI allows sequencing of algorithms in which an arbitrary number of algorithms can be ran with input coming from sensed data or another algorithms output. After each selection of an output, the UI updates the possible input options for other algorithms. The raw data is represented as RAW and the final output is represented as OUTPUT. The Network window employs the same design where a network consists of a directional link between two communication components of Mote and Base station.



Mote Configuration
Mote ID: heartMote
Platform: shimmer2r
Sensors: ECG

Communication Configuration
Platform: ZigBee / Bluetooth

Energy Management Configuration
Radio: Radio ON
Duty Cycling: Dynamic
Radio Power Level: 3
Duty Cycle Properties
Wakeup Time: 1000
Sleep Time: 1500
Minimum Power Level: 3
Maximum Power Level: 23

Sensor Configuration View
Sensor Type: ECG
Sampling Frequency: 512 Hz
Sample Size: 300
Algorithm: Off
Algorithms: hd_peakdetect
hd_fft
hd_peakdetect

Algorithm Configuration
peakdetect — INPUT — peakdetect — OUTPUT: OUTPU
fft — INPUT: RAW — fft — OUTPUT: peakde

Network Configuration
Name: WHS
heartMote -> smartphone
heartMote
phone
smartphone

BaseStation Configuration
BaseStation ID: smartphone
Platform: Android
Communication Configuration
Protocol: Bluetooth
ZigBee
Bluetooth

Figure 4.2: Health-Dev User Interface.

The GUI is designed to abstract the AADL syntax and constructs into a series of windows. It represents each AADL construct as a component. The components can be Mote, Sensor, Communication, Energy management, Algorithms, Base station, and Network. For each component, there exists a window dedicated to configuring just that component. Components can be nested e.g. a sensor in a PHMS has no value without existing in a mote and mote has no way to communicate without communication component. Figure 4.2 shows nesting of components. One of the effective method to design such GUI is to follow software design pattern called Model-View-Controller in which a model stores the data in a logical fashion, a view represents the data using an interface and controller responds to the user input by modifying the model. It acts as a mediator between model and view. In the following, each component of GUI is discussed.

Mote component

In mote component user specifies the data computation flow, communication and energy management requirements for each type of sensed signal. Based on the hardware capability of a sensor platform, multiple sensors can be implemented for the same. The component allows user to specify identifier, types of sensor, e.g. ECG or accelerometer or temperature or humidity, sampling frequency, sensitivity and platform type. Currently, *Health-Dev* supports commercially available TinyOS-based platform such as Telosb (xbow.com), Shimmer (shimmer-research.info), BSN v3 (http://vip.doc.ic.ac.uk/bsn/a1892.html) and Linux based platform such as iMote2 (bullseye.xbow.com) and Arduino (arduino.cc). In communication subcomponent, user can specify communication protocol, which can be either Bluetooth or ZigBee. The computation subcomponent allows specification and sequencing of physiological algorithms available in the code base. The sequencing determines the flow of sensed signal through series of algorithm. Due to power constraint in the battery operated sensor platform, energy and reliability management subcomponent allows user to specify duty cycles and dynamic power management schemes to optimize power consumption. The figure 4.3 summarizes the properties a user can specify in Mote component.

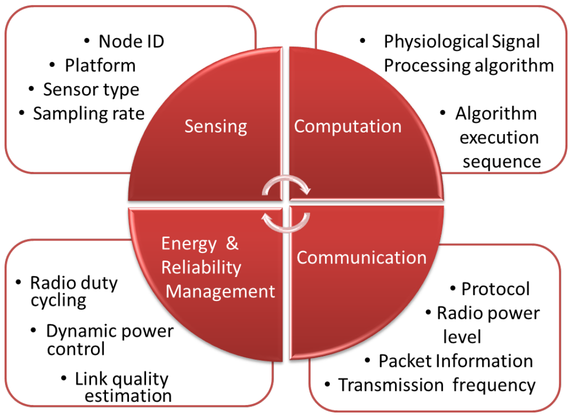


Figure 4.3: Specifiable properties in Mote component.

Base station component

The base station component can be either a personal computer (PC) or a smart phone. User can select the communication protocol which is either ZigBee or Bluetooth based upon the hardware capability of the selected platform. Currently, *Health-Dev* supports only Android smart phones with Bluetooth capability and PC with ZigBee. For smart phone, *Health-Dev* provides a basic data visualization interface that allows starting and stopping of graphing event and running basic algorithms such as computation of standard deviation of received data. In addition, standard widget buttons provided can be used to control the sensing properties of a specific sensor.

Network component

Network component allows user to link sensor component with Base station to establish a network. This high-level approach to implement PHMS provides user with a better understanding of the excepted outcome of the implementation.

4.2 Parser module

Parser module takes the AADL model as an input and generates meta-models, which are used by code generator to generate sensor and smart phone code for the implementation. One of the main function of parser is to group all the sensor specifications based on similar node id and sensor platform. For a given algorithm specification of a sensor, parser matches appropriate algorithm filename from the platform specific codebase and includes it in meta-model. It then parses the connections in the specification and for each algorithm extracts the input map. Similarly, for smart phone specification, parser is concerned about setting data visualization interface and algorithm specification.

4.3 Code generation module

The meta-models goes as an input to code generator which outputs sensor and smart phone code. As the generator reads through meta-models, it calls platform specific code templates and generates module file to enable required sensor properties and communication protocol by using pre-processing directives. If algorithm is specified in meta-model, generator will pull the required algorithm from codebase to declare, provide sequencing with sensed raw data and injects it to specific location in the code. Similarly a wiring file is generated with all possible hardware modules differentiated by pre-processing directives. In smart phone code generation, code generator uses the Application Programming Interfaces (APIs) provided by *Smart phone Middleware* to ensure better data handling when smart phone receives the incoming data. *Smart phone Middleware* allows communication between external wireless mote and an Android smart phone via Bluetooth by using set of APIs. It consists of two components:

Bluetooth API

The main function of this component is to ensure connection establishment and data communication between mote and smart phone. This functionality is broken down into parts.

- The smart phone provides user with a GUI containing a list of external mote to connect with. On selecting a particular mote, Bluetooth API utilizes Android Bluetooth API to perform scan of nearby Bluetooth enabled devices to connect with the selected mote.

- Once the connection is established, a new screen pops up allowing user to control the external mote by start, stop, and updating the sampling frequency of the sensor on-the-fly.

After sending start sensing command, Bluetooth API prepares itself to receive incoming data from the mote. The data is received by smart phone in customized packet format is parsed by Bluetooth API to extract the content before it is sent to Sensor Handler. User is provided with a range of sampling frequency which can be exploited depending upon criticality of physiological data and battery usage of the smart phone.

Sensor Handler

This component acts as a manager; registering all user assigned sensor, different algorithms associated with each sensor and handling of data received from particular sensor of a connected mote. As each mote may have more than one sensor, Sensor handler maps the incoming data to a specific sensor to allow correct data processing. Sensor handler stores the sensor type and name of the mote to eliminate any re-instantiation of the same in future. As Bluetooth API forwards the data to Sensor handler, it calls the instance of current mote with current sensor type to process the data. Once the data is processed it is returned back to sensor handler to update the smart phone UI with new data.

Chapter 5

EVALUATION, VALIDATION and USE CASES

As software plays a major in safer operation of PHMS, there is a need to have a metric to measure the quality and efficiency of the generated code for sensor and smart phone for a personalized health monitoring system. In this regard, code size, RAM usage, safe memory access, race conditions, optimization and energy consumption were considered while comparing *Health-Dev* code with BSNBench [7], benchmark code for Body Sensor Networks (BSNs). Since TinyOS is a popular operating system designed for low-power wireless sensor networks, evaluation of TinyOS code generated by *Health-Dev* against manually written BSNBench code were considered. In addition, two experiments are discussed as a validation of generated code satisfying the requirement set in design phase.

5.1 Evaluation

The generated code was evaluated using 4 methods shown shows in the Figure 5.1:

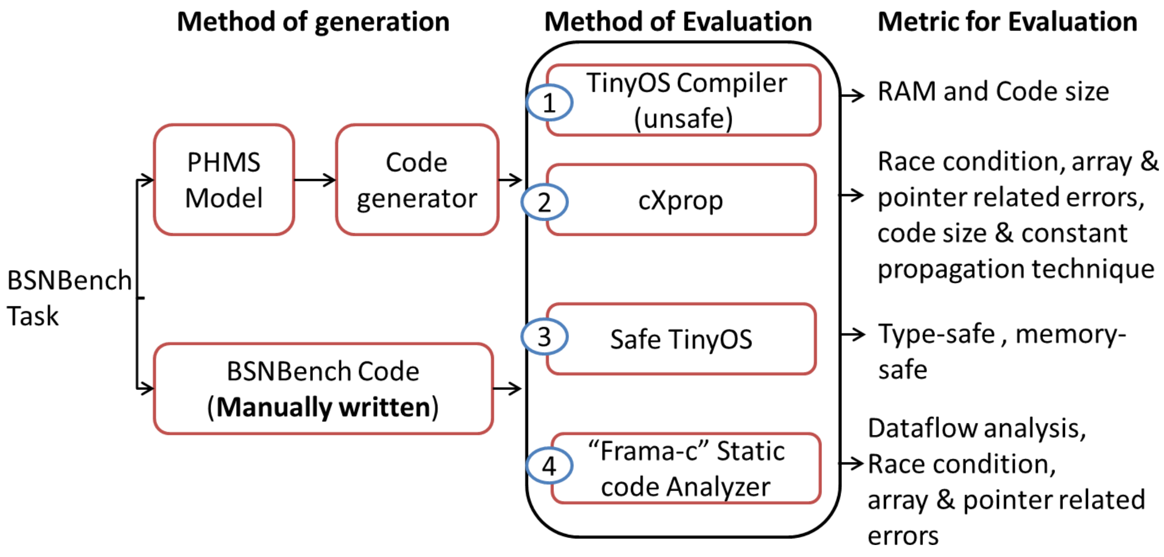

Method of generation
Method of Evaluation
Metric for Evaluation
BSNBench Task
PHMS Model
Code generator
BSNBench Code (Manually written)
1 TinyOS Compiler (unsafe)
2 cXprop
3 Safe TinyOS
4 "Frama-c" Static code Analyzer
RAM and Code size
Race condition, array & pointer related errors, code size & constant propagation technique
Type-safe , memory-safe
Dataflow analysis, Race condition, array & pointer related errors

Figure 5.1: Methods of Evaluation.

- TinyOS Compiler: It is a widely used compiler for TinyOS applications but often regarded as unsafe because it does not checks for array and pointer related errors which can corrupt memory during runtime operation of PHMS software.

- cXprop: cXprop is static analyzer and source to source optimizer for embedded C programs such as nesC [6]. cXprop implements conditional X propagation, a generalization of the conditional constant propagation algorithm where X is an abstract value domain supplied by the user. It is implemented on CIL (C Intermediate Language) [24] which acts as a parser, type-checker, and intermediate representation for C. cXprop tracks data-flow in values through scalars, pointers, structure fields, arrays, and global scope.

- Safe TinyOS: Safe TinyOS is build on the top of TinyOS, ensures type and memory safety during execution [6]. It uses cXprop to optimize the code. In Safe TinyOS tool chain uses Deputy [25, 26], a source-to-source compiler for ensuring type and memory safety in C code, to modify the nesC source files by including annotations. These modified source files and translated to C code by modified nesC compiler. The tool chain involves four source-to-source transformation steps as mentioned in [6]. First, safety check are added by Deputy compiler. Second, ensuring safety checks are maintained under interrupt driven concurrency. Third, compress safety violations information and fourth, perform whole program optimization by cXprop.

- ”Frama-c” (static code analyzer): Frama-c analyses the code using forward data-flow analysis based on the principles of abstract representation [27]. It checks for array and pointer related errors in the code.

Table 5.1: Comparison of RAM and Code size of generated code for TinyOS-based sensor.

Task	**Auto-generated Code RAM Size**	**BSNBench RAM Size**	**Auto-generated Code Size**	**BSNBench Code Size**
FFT	3541	4403	19342	17430
Peak detect	1243	1279	20068	14488
FIR	484	1075	17490	11582
Differential encoding	4443	4275	17538	11512
Out of range	1241	1087	18478	13076
Statistics	4441	2285	17084	14270

Table 5.1 shows the comparison between *Health-Dev* code and BSNBench code [7]. It shows that code size of *Health-Dev* is always greater than the BSNBench. This is because *Health-Dev* aims to develop generic codes that can be easily modified to serve a different application and BSNBench code is optimized for a specific functionality. However, *Health-Dev* has lower RAM size than BSNBench as *Health-Dev* variables used in physiological signal processing algorithms (FFT, peak detection, and FIR filter) are declared within the function to make them modular resulting in lesser global variables. Where as in BSNBench code, internal variables of a function are declared as global variable an reused to optimize the RAM usage. In case of data operation algorithm the RAM size of generated code increases due to the fact that the algorithms were implemented as Task and the variables associated with the algorithm were of larger data type. The energy consumption for the *Health-Dev* generated code was found similar to that of BSNBench code, with variances attributed to battery power fluctuations and measurement errors.

Application code running in PHMS may not be safe due to presence of array and pointer related errors as well as race conditions . Safer execution of application code can ensure ar-

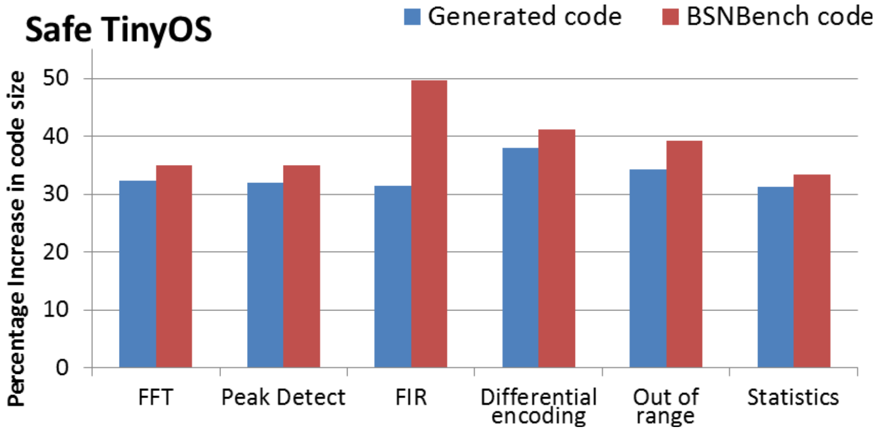


Figure 5.2: Code size comparison between generated and BSNBench code after Safe TinyOS.

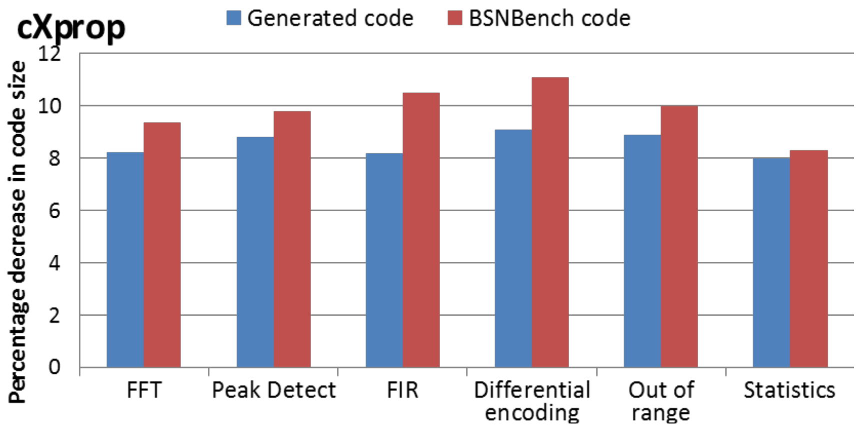


Figure 5.3: Code size comparison between generated and BSNBench code after cXprop.

ray and pointer related errors are caught before they corrupt the RAM during run-time [6]. *Health-Dev* code were compared with BSNBench code in terms of safety and optimization in code using Safe TinyOS [6] and cXprop [6]. The Figure 5.2 and Figure 5.3 shows percentage change in the code size observed when Safe TinyOS and cXprop were applied on code generated automatically and manually. The change code size observed with respect to the code of size found using ordinary TinyOS tool chain. In each case there is an increase in the code size when Safe TinyOS is applied. This is because Safe TinyOS inserts safety annotations in the code and stores error messages in ROM (stores code) of a micro-controller for safer run-time operation. However, cXprop used in Safe TinyOS reduces average code size cost. In order to include safety checks code size is increased and can be seen as a trade-off between safety and code size. Whereas cXprop only performs optimization without safety checks and hence the code size is found to be reduced. It was inferred, if the change in code size is higher, it indicates that more optimization is performed. Hence, code with lesser change in code size are more optimized. Generated code (FIR algorithm) was found to be 9.3% more optimized than manually written code. When ”Frama-c” was applied to the generated code, it was observed that no error related to array (out-of-bounds) and pointers (bad pointer access) were found. Hence, ensuring safety and optimal use of memory of the micro-controller.

Most of the embedded micro-controllers including TI MSP430 lacks memory protection and it makes the use of dynamic memory allocation risky as it may incur race conditions. *Health-Dev* ensures the variables in the generated code don't incur race condition and makes the code safe from incurring overlapping of heap and stack segment of a memory.

5.2 VALIDATION

Two case studies are discussed and showed that *Health-Dev* generated code satisfies energy requirement for a give processing operation by using radio duty cycle and provides mobility aware transmission of packet by changing the transmission power based on link quality.

Radio duty cycling

A typical PHMS has to operate in an unsupervised environment for longer duration of time, sensor of PHMS must be supported by multiple power sources. Considering human body heat can generate upto 150 mW of power for 6 hrs [12] and if that energy is used to power a sensor for 24 hrs, then maximum power consumption by sensor must not be more than 37.5 mW each hour. Having this constraint as a requirement radio duty cycle was calculated as 8.2% using the following equation 5.1, where at any give time t and duty cycle $x\%$, the energy consumption by sensor platform (E_{sensor}) was given by-

$$E_{sensor} = ((P_{ROn} + P_{proc})\frac{x}{100} + P_{proc}\frac{1-x}{100}) \times t. \tag{5.1}$$

power consumption of the sensor platform was profiled as radio on ($P_{ROn} + P_{proc}$) and radio off (P_{proc} only computation power) [7].

In this case study, a PHMS specification where wireless sensor was considered to sense ECG signal, performing peak detection, and FFT, ECG based signal processing algorithms [7] were specified in AADL specification of *Health-Dev* with the same radio duty cycle as 8.2% and the generated code was installed in TelosB mote platform. The average power consumption, measured over a single operation cycle, was 10.84 mW, which was much lesser than the average power available from scavenging sources (150 6/24 = 37mW). Hence, the set forth requirements was achieved.

Mobility aware transmission control

Under dynamic context, mobility of body incurs considerable variation in Packet Error Rate (PER) which is the ratio of total packets received to total packet sent. This case study aimed to reduce PER by sensor estimating the link quality in terms of path loss due to fading on each transmission. Link quality between pair of node (A,B) gives the probability that a packet transmitted by A will be successfully received by B [28] as it maintains close correlation with Packet Received Ratio (PRR). For CC2420 radio, commonly used transceiver in wireless sensor platforms, the value of link quality varies from 50 to 110 where 50 indicates minimum value (worst quality) and 100 represents maximum (best quality). If the path loss is lesser than a threshold value then transmission power of the radio is increased else decreased to lowest power level. PHMS with this power control schedule was specified in AADL. In model based analysis phase, the Ricean flat fading was assumed with Bit Error Rate (BER) as a function of path loss [29] as suggested by IEEE Task Group 6.

As CC2420 has eight levels of power ranging from -25dBm to 0dBm, the path loss ranged from 10 dB to 70 dB. For a given BER, PER was calculated using-

$$PER = 1 - (1 - BER)^L. \tag{5.2}$$

where L is the packet length. To perform the experiment Levi walk model[ref] was used considered for mobility of a human subject. It was found that at lowest radio power transmission (-25 dBm), worst case of PER was observed as 0.82, whereas a transmit power of -15 dBm gave worst case PER of 0.05. Outdoor excursion probabilities, under Levi walk model ranging from 0.2 to 0.9, introduced PER of 0.12, during alternating power transmission between -25 dBm and -15 dBm. For a reliable network operation a PER of 0.12 was considered. The resulting PHMS model was passed through *Health-Dev* to generate the code. The experiments were performed in campus and outdoor movements were simulated by short excursions outside the department office. The PER was found to be 0.03, which satisfied the requirement set in design phase.

5.3 USE CASE

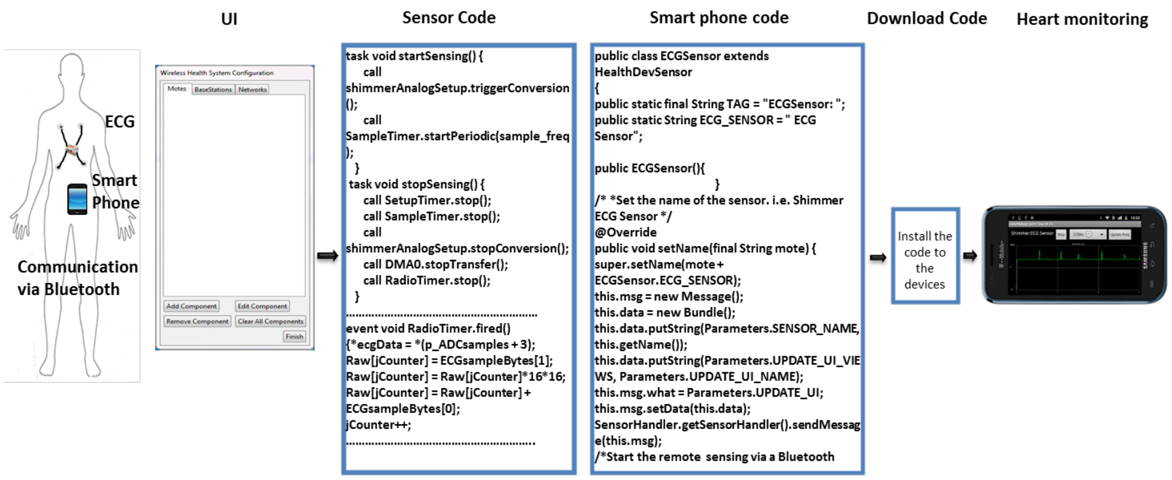
Two use cases of model-based code generator used by *Health-Dev* are showcased, providing an ease of implementation of continuous health monitoring system and a way of facilitating the access of physiological data to an existing Android App.

Continuous Monitoring System

The objective of this use case was to showcase the ability of *Health-Dev* to implement personalized health monitoring system. Implementation of continuous ECG monitoring system without worrying about low-levels of details was considered. In order to generate code for such system *Health-Dev* allowed user to interact with a GUI to configure the system at a high-level as shown in the Table5.2.

Table 5.2: Specifiable properties for Case Study I.

Properties	Mote configuration	Base station configuration
Platform	Shimmer	Android
Communication Protocol	Bluetooth	Bluetooth
Sensing type	ECG	-
Sampling frequency	256 Hz	-
Radio	on	on



```
task void startSensing() {
    call
shimmerAnalogSetup.triggerConversion
();
    call
SampleTimer.startPeriodic(sample_freq
);
  }
 task void stopSensing() {
    call SetupTimer.stop();
    call SampleTimer.stop();
    call
shimmerAnalogSetup.stopConversion();
    call DMA0.stopTransfer();
    call RadioTimer.stop();
  }
.........................................................
event void RadioTimer.fired()
{*ecgData = *(p_ADCsamples + 3);
Raw[jCounter] = ECGsampleBytes[1];
Raw[jCounter] = Raw[jCounter]*16*16;
Raw[jCounter] = Raw[jCounter] +
ECGsampleBytes[0];
jCounter++;
.........................................................
```

```
public class ECGSensor extends
HealthDevSensor
{
public static final String TAG = "ECGSensor: ";
public static String ECG_SENSOR = " ECG
Sensor";

public ECGSensor(){
                                }
/* *Set the name of the sensor. i.e. Shimmer
ECG Sensor */
@Override
public void setName(final String mote) {
super.setName(mote +
ECGSensor.ECG_SENSOR);
this.msg = new Message();
this.data = new Bundle();
this.data.putString(Parameters.SENSOR_NAME,
this.getName());
this.data.putString(Parameters.UPDATE_UI_VIE
WS, Parameters.UPDATE_UI_NAME);
this.msg.what = Parameters.UPDATE_UI;
this.msg.setData(this.data);
SensorHandler.getSensorHandler().sendMessag
e(this.msg);
/*Start the remote  sensing via a Bluetooth
```

Figure 5.4: Design flow of Continuous Monitoring System.

The Figure 5.4 gives the flow of the implementation where a high-level specification of continuous monitoring system was converted to sensor and smart phone code. The generated code was automatically downloaded to respective devices. User could change the sensor property like sampling frequency and Start/Stop the sensor remotely. Similarly, continuous motion monitoring of motion can also be implemented by selecting accelerometer as sensing type.

Table 5.3: Specifiable properties for Case Study II.

Properties	**Mote configuration**	**Base station configuration**
Platform	Shimmer	Android
Communication Protocol	Bluetooth	Bluetooth
Algorithm	Heart-rate	-
Sensing type	ECG	-
Sampling frequency	256 Hz	-
Radio	on	on

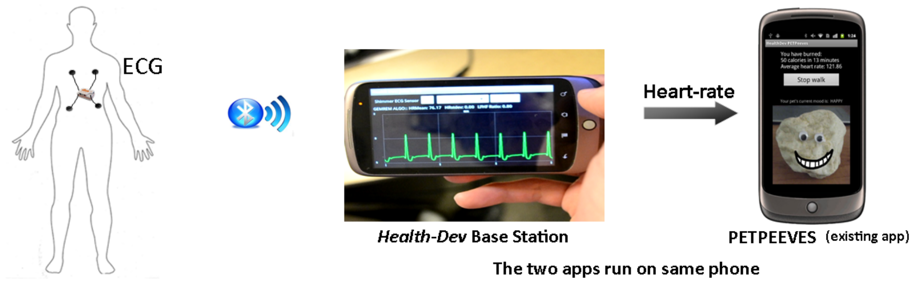


Figure 5.5: Health-Dev base station forwarding heart rate data to PetPeeves.

Physiological data facilitator to smart phone app

This use case showcased an Android App called *PetPeeves* utilizing physiological data provided by *Health-Dev* Base station application. The Base station acts as a physiological data facilitator to *PetPeeves* which promotes exercise in a fun way. PetPeeves is a virtual pet based app where user has to keep the pet happy by doing exercise. The facial expression of the pet is proportional to how much a user exercises. It also calculates calories burned from heart-rate and tracks motion of the person via Global Positioning System (GPS). To design a system where sensor mote senses ECG signal, calculates heart-rate and transmit the data via Bluetooth to *PetPeeves* via *Health-Dev* base station application, these were the following information required by GUI mention in Table 5.3.Once this system is deployed, during runtime the physiological data received by *Health-Dev* Base station was forwarded to Pet Peeves where the pet will react based upon heart-rate value which is proportional to exercise performed by the user. Figure5.5 gives the overview this use case. It encourages app developers to create health-based smart phone apps without worrying about computing physiological information from the sensor.

Chapter 6

CONCLUSION AND FUTURE WORK

In this thesis work a model-based automatic code generator for PHMS was developed, generated code were evaluated and validated against the set requirements. The generated code were experimentally evaluated and validated in terms of memory access and logic implementation. It was found that generated code was 9.3% more optimized as compared to BSNBench code in terms of less unreachable code and ensured safer operation of PHMS software by avoiding race conditions, arrays and pointers related errors from unintentional exploitation of already resource constraint memory of embedded micro-controller. Static code analyzer on generated code resulted with no race conditions nor arrays or pointers related errors. Areas of future works can be classified as follows:

- Guaranteeing safety in TinyOS-based sensor incurs costs in terms of source-code modification and sensor hardware resources. Code generator for PHMS software must ensure very less source-code modification when Safe TinyOS [6] is applied. One of the approach is to ensure that platform-specific template code is optimized for memory efficient usage.

- Safety in Arduino-based sensor : Due to increase in the usage of Arduino sensor hardware for PHMS [30], safety hazards of Arduino programming (C++-based language) should be inspected to ensure safety in sensor code.

- Extension of platform support : Newer hardware platforms are emerging for wearable medical sensors such as Electroencephalography (EEG), Electromyography (EMG), Galvanic Skin Response (GSR) and smart phones (Windows and BlackBerry) which run on different software, there is a possibility to extend the support for such platforms with sensor specific signal processing algorithms in the code generator.

- Signal processing algorithms : Physiological signal processing algorithms may require to have adaptive thresholding to respond to sudden change in human physiology while keeping energy consumption of sensor within a limit. Hence, development of such algorithms would enhance the ability of code generator.
- Web-based interface : A web-interface to the code generator would allow others in the research community to use it for their purpose.

REFERENCES

[1] K. Venkatasubramanian, G. Deng, T. Mukherjee, J. Quintero, V. Annamalai, and S. Gupta, “Ayushman: A wireless sensor network based health monitoring infrastructure and testbed,” in *Distributed Computing in Sensor Systems* (V. Prasanna, S. Iyengar, P. Spirakis, and M. Welsh, eds.), vol. 3560 of *Lecture Notes in Computer Science*, pp. 406–407, Springer Berlin Heidelberg, 2005.

[2] http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/ HumanFactors/UCM290561.pdf.

[3] FDA. http://www.accessdata.fda.gov/.

[4] D. R. Wallace and D. R. Kuhn, “Failure modes in medical device software: an analysis of 15 years of recall data,” in *IEEE International Conference on Computer Systems and Applications*, pp. 301–311, 2001.

[5] TinyOS. http://en.wikipedia.org/wiki/TinyOS, 2013.

[6] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, “Efficient memory safety for tinyos,” in *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys ’07, (New York, NY, USA), pp. 205–218, ACM, 2007.

[7] S. Nabar, A. Banerjee, S. K. S. Gupta, and R. Poovendran, “Evaluation of body sensor network platforms: a design space and benchmarking analysis,” in *Wireless Health 2010*, WH ’10, (New York, NY, USA), pp. 118–127, ACM, 2010.

[8] http://www.doc.ic.ac.uk/vip/ubimon/bsn_node/index.html.

[9] Shimmer-Research. http://www.shimmer−research.com/, 2013.

[10] http://www.fda.gov/downloads/AboutFDA/CentersOffices/OfficeofMedical ProductsandTobacco/CDRH/CDRHReports/UCM308208.pdf.

[11] http://www.fda.gov/MedicalDevices/Safety/AlertsandNotices/Tipsand ArticlesonDeviceSafety/ucm070185.htm.

[12] A. Banerjee, S. Kandula, T. Mukherjee, and S. K. Gupta, “BAND-AiDe: A tool for cyber-physical oriented analysis and design of body area networks and devices,” *ACM*

Trans. on Embedded Computing Systems, Special issue on Wireless Health Systems, 2010.

[13] M. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri, "A framework for modeling, simulation and automatic code generation of sensor network application," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pp. 515 –522, june 2008.

[14] V. M. Jones, A. Rensink, T. C. Ruys, H. Brinksma, and A. T. van Halteren, "A formal mda approach for mobile health systems," in *EWMDA-2, Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations, Canterbury, England* (D. H. Akehurst, ed.), no. TR 17-04, (Canterbury), pp. 28–35, University of Kent, September 2004.

[15] S. Glesner, R. G. B. Boesler, R. Gei, and B. Boesler, "Verified code generation for embedded systems," 2002.

[16] T. Moreira, M. Wehrmeister, C. Pereira, J.-F. Petin, and E. Levrat, "Automatic code generation for embedded systems: From uml specifications to vhdl code," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pp. 1085 – 1090, july 2010.

[17] Z. Shu, D. Li, Y. Hu, F. Ye, S. Xiao, and J. Wan, "From models to code: Automatic development process for embedded control system," in *Networking, Sensing and Control, 2008. ICNSC 2008. IEEE International Conference on*, pp. 660–665, april 2008.

[18] V. Prasad et al, "Andes: An analysis-based design tool for wireless sensor networks," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pp. 203–213, Dec. 2007.

[19] C. Bunse, H.-G. Gross, and C. Peper, "Applying a model-based approach for embedded system development," in *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pp. 121–128, Aug. 2007.

[20] J. Cuadrado and J. Molina, "A model-based approach to families of embedded domain-specific languages," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 825 –840, nov.-dec. 2009.

[21] A. Ghosh, Y.-K. Hui, and M. Chiang, “Model-based architecture analysis for wireless healthcare,” in *Proceedings of the First ACM MobiHoc Workshop on Pervasive Wireless Healthcare*, MobileHealth ’11, (New York, NY, USA), pp. 12:1–12:4, ACM, 2011.

[22] AADL. http://www.aadl.info/aadl/currentsite/start/embedded.html, 2013.

[23] M. Voelter, “A catalog of patterns for program generation,” in *European Conference on Pattern Languages of Programs 2003*, 14 2003-april 2003.

[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of c programs,” in *Proceedings of the 11th International Conference on Compiler Construction*, CC ’02, (London, UK, UK), pp. 213–228, Springer-Verlag, 2002.

[25] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, “Dependent types for low-level programming,” in *In European Symposium on Programming*, 2007.

[26] Deputy. http://ivy.cs.berkeley.edu/ivywiki/uploads/deputy-manual.html.

[27] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Los Angeles, California), pp. 238–252, ACM Press, New York, NY, 1977.

[28] L. Estimation. http://www.tinyos.net/tinyos-2.x/doc/html/tep124.html.

[29] I. T. group. http://math.nist.gov/mcsd/savg/papers/15-08-0780-09-0006-tg6-channel-model.pdf.

[30] Arduino. http://www.arduino.cc/, 2013.