ABSTRACT

Title of Document: SYNTHESIS OF EMBEDDED SOFTWARE FOR SENSOR NODES

Celine Badr, Master of Science, 2006

Directed by: Professor Shuvra S. Bhattacharyya, Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies

In this work, we address the synthesis of embedded software for sensor nodes in two important, specialized contexts. In the first context, an optimization framework is designed to automate the design space exploration of application-specific wireless sensor networks in order to adjust configuration parameters for deriving a streamlined overall implementation of the system. The framework is built around the particle swarm optimization technique and adapted especially for the optimization of a line-crossing detection application.

The second synthesis context draws from the potential effectiveness of using dataflow graphs for the implementation of DSP applications for sensor nodes to explore a context switching mechanism facilitating concurrent execution of multiple dataflow graphs on a single embedded processor. Our model for context switch implementation uses compile-time information to optimize runtime scheduling. Simulation results in both cases support the applicability of the adopted approaches for optimized operation of application-specific sensor nodes.

SYNTHESIS OF EMBEDDED SOFTWARE FOR SENSOR NODES

By

Celine Badr

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of

Master of Science

2006

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chair

Professor Neil Goldsman

Professor Gang Qu

Dedication

To my family.

Acknowledgements

I would like to thank my advisor, Professor Shuvra S. Bhattacharyya, for his continued and valuable guidance. It was a pleasure for me to be part of the DSPCAD group and to interact and work with all the members of this research group. I would particularly like to express my gratitude to Chung-Ching Shen for his important contribution and assistance in parts of this work, and to Dong-Ik Ko and Chia-Jui Hsu for their input related to the dataflow context switching project. I also wish to thank my defense committee, Professor Neil Goldsman and Professor Gang Qu, for their insightful suggestions, as well as Professor Gilmer Blankenship and Kamiar Kordari for providing the specifications of the line crossing detection application. Portions of this research (supporting materials, equipment, etc.) were supported by the Advanced Sensors Collaborative Technology Alliance, and the University of Maryland Smart Dust Research Project, which is sponsored by the Department of Defense, Maryland Procurement Office (Contract #H9823004C0490).

A word of thanks also goes to Marc Ó Donnagáin and Ivan Corretjer for their patient proofreading.

Finally, I would like to thank all my relatives and friends for their care, their prayers and their continuous encouragement, and most importantly, I want to express my thanks to my parents and my brother in Lebanon for their love and their unconditional support.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

With the recent emergence of diverse applications for wireless sensor nodes, a large amount of literature is addressing specific needs and features of those applications, such as power consumption, latency, transmission protocols, network usage and others. In order to demonstrate any developed sensor network applications on real hardware platforms for the sensor nodes, users usually need to design experimental prototype platforms for wireless sensor network systems by drawing from an increasing variety of off-the-shelf hardware and software components. Such components are often reconfigurable across a range of settings to allow users to tune the functionality and associated implementation trade-offs. Separate efforts were put into trying to optimize individual metrics in the context of specific projects. However, given the stated wireless sensor network systems dependence on numerous inter-related parameters, the associated design space is vast, and effective optimization in this space is challenging due to the combinatorial growth of admissible system configurations and to the complexity of interactions among component and system parameters.

In the first part of this work, we introduce a system-level design methodology to find efficient configurations for an application-specific sensor network system where optimization of energy consumption is a primary implementation criterion. By analyzing critical parameters from candidate off-the-shelf components that may be employed in construction of the network, and integrating the associated parameters into a comprehensive optimization framework, we automate the exploration of the design space. The optimized configurations derived from our optimization framework can then be applied to the actual hardware implementation of the targeted sensor network system.

The optimization framework is implemented as a Java-based software tool, built on the particle swarm optimization strategy, which is extended to meet the requirements of our target application. Specifically, the framework is adapted and tested for the line crossing detection application.

The second part of our work relates to another synthesis context for sensor nodes. Motivated by the potential effectiveness of using dataflow graphs for the implementation of DSP applications for sensor nodes, we introduce a context switching mechanism and explore its applicability to dataflow programming in this perspective. Considering the size, power, real-time operation, and other constraints faced by embedded applications in DSP, the utility of specialized computational models such as dataflow graphs has become a well-known fact. It draws mainly from their intuitive specification mechanism for DSP and their ability to expose relevant application structures and parallelism at compile time. This facilitates the exploitation of special application characteristics and simplifies static scheduling techniques. Our model for context switch implementation uses such compile-time information to optimize a runtime scheduling operation that handles concurrent execution of multiple graphs.

The rest of this document is organized as follows: Chapter 2 details the design, implementation, and simulation of the particle swarm optimization-based framework for application-specific sensor networks, as adapted to the line crossing detection application. Chapter 3 introduces the role of dataflow graphs in DSP, and describes the proposed context switching model and its scheduling constituents. Algorithms, implementation aspects, and application prototyping are also presented. Finally, conclusions and future directions are listed in Chapter 4.

Chapter 2: Optimization Framework for Application-Specific Sensor Networks

We describe in this chapter the design and implementation procedures of the optimization framework for application-specific sensor networks. The chapter also includes background information about the particle swarm optimization technique and some related work in the areas of sensor network modeling and optimization.

2.1 Overview

Wireless sensor network (WSN) nodes generally combine four subsystems under constraints of limited hardware resources, very small size, and low cost. These subsystems are for computation, communication, sensing, and power [2][16][20]. Users can design experimental prototype platforms for WSN systems by drawing from an increasing variety of off-the-shelf hardware and software components. Such components are often reconfigurable across a range of settings to allow users to tune the functionality and associated implementation trade-offs. Such reconfigurable components provide different parameters that must be carefully adjusted to derive streamlined implementations for specific WSN applications. The design space associated with the optimization of WSN configuration is vast due to the combinatorial growth of admissible system configurations and the complexity of interactions among component and system parameters. For efficient design space exploration in this challenging context, we have developed a Java-based software tool for careful modeling and extensive optimization of sensor network components along with their associated configuration options.

The optimization approach in our tool is based on a general strategy called particle swarm optimization (PSO), which was introduced in [15] as a population-based, optimization technique for simulating the social behavior of individuals. The PSO strategy explores a problem space with an evolving set of candidate solutions, which are referred to as particles. The set of particles under consideration at any given time is called the swarm, and this swarm is "flown" through the D-dimensional search space through various manipulations to find an optimized solution.

In the context of our optimization framework, we have developed a generic PSO package along with novel plug-ins to this package that provide customized features for WSN-related optimization. Given a user defined sensor network, mutable and immutable parameters for configuring components and system parameters in the network, and application-specific models for evaluating the relevant design evaluation metrics as a function of the network configuration, our optimization framework derives an efficient WSN configuration that is optimized for minimum energy consumption. Due to the accuracy of the evaluation methods employed in our optimization framework, the effectiveness of the optimization approach, and thoroughness with which configurations are managed, solutions derived from the framework can be mapped efficiently into hardware/software implementations of complete, application-specific WSN systems.

2.2 Related Work

Various research groups have built sensor node platform with interesting combinations of features (e.g., see [1][9][16][18]). These approaches generally involve

off-the-shelf components, and include detailed measurement of power consumption or performance analysis from the constructed platforms. However, few such works are integrated with strategies for system-level modeling and optimization.

Akyildiz et al. [2] provide a comprehensive survey on applications, design factors and communication architectures for wireless sensor networks, including elaboration on the physical constraints on sensor nodes and protocols proposed in all network layers.

Singh et al. [23] discuss system-level trade-offs related to energy costs of state-of-art WSN technologies. An integrated, system-level model and energy trade-off analysis is presented for application development in wireless networks. This work also presents a system-level energy model that incorporates the energy consumption associated with computation, storage, communication, and sensing. A model-based integrated simulation tool called MILAN is introduced as well, which facilitates rapid, multi-granular evaluation of energy consumption and performance for a large class of systems. A major feature of MILAN is that it integrates a variety of different simulators and design tools into a unified environment.

Jin et al. [13] discuss an approach to sensor network optimization using a genetic algorithm. In this paper, the authors attempt to systematically cluster groups of WSN nodes to exploit the advantages of small transmission distances. They also discuss how clustering is a NP hard problem, and outline a method to determine an efficient selection of cluster heads using a genetic algorithm approach.

The technique of particle swarm optimization (PSO) [15] has been the subject of extensive research in recent years. Due to the simplicity of its implementation and the

small number of parameters involved in its fine tuning, PSO has been used to solve optimization problems in a wide variety of applications. A summary of the main developments and applications pertaining to PSO can be found in [6].

Multi-objective optimization using PSO is addressed in [12] by implementing a dynamic neighborhood version that optimizes one objective function while fixing the other, and in [19] by studying weighted aggregation in addition to an approach that exchanges information among multiple swarms.

For wireless sensor networks, the PSO approach has been adopted in [27] to optimize clustering techniques. Another example is [25] where it is demonstrated that a PSO-based approach converges faster than a GA-based approach in finding energy-minimized WSN clusters by localizing cluster-head nodes. Veeramachaneni et al. [28] describe a PSO-based strategy to dynamically configure sensor thresholds and their related fusion rules in a biometric sensor network, while considering both accuracy and time.

Our work differs from these approaches in that our method is not specific to a particular WSN application or to a particular configuration problem such as clustering. Our approach aims to provide a more general methodology and associated computer-aided design tool for taking into account arbitrary combinations of WSN network configuration parameters and their associated interactions. The trade-offs between one set of possible configuration parameters and another with regards to the optimization objectives are evaluated within the framework in an automated process.

2.3 Optimization Framework Preliminary Design

Given some application specifications and architectural assumptions, the optimization framework is expected to generate a set of system configurations that implement the application optimally with respect to certain evaluation metrics. This is possible by identifying two main components of our framework as shown in Figure 1: a design space and an evolutionary space.

2.3.1 Design Space

The design space takes the application specifications and the architectural assumptions as input and identifies possible configurations for consideration in our optimization. The possible configurations are composed of candidate components for the different entities that constitute our system along with instantiated parameter sets.

We consider the fact that some candidate components are to be fixed in our system frame (immutable CCs), whereas others offer different variations that can be potentially integrated after evaluation (mutable CCs).

Also, regarding the parameters, some values are accepted as immutable because of application constraints, designer choice, or hardware limitations, whereas other parameters can assume various values from defined domains.

The mutable entities are encoded to constitute a representation specific to a configuration instance. On the other hand, immutable CCs and parameters can be directly sent as input to an evolutionary space subsystem, specifically to the scheduling analyzer.

2.3.2 Evolutionary Space

The evolutionary space is mainly an iterative system that considers a present configuration (population) to implement our application. This implementation is then evaluated by applying objective functions relative to our targeted metrics. Based on the fitness of the solution, evolutionary techniques can be applied. Also, local heuristics can be introduced such as a random or preset weighing of the metrics. Then the new population thus obtained is evaluated, and the cycle repeats until the controller decides on the configuration or set of configurations that will be output as its solution.

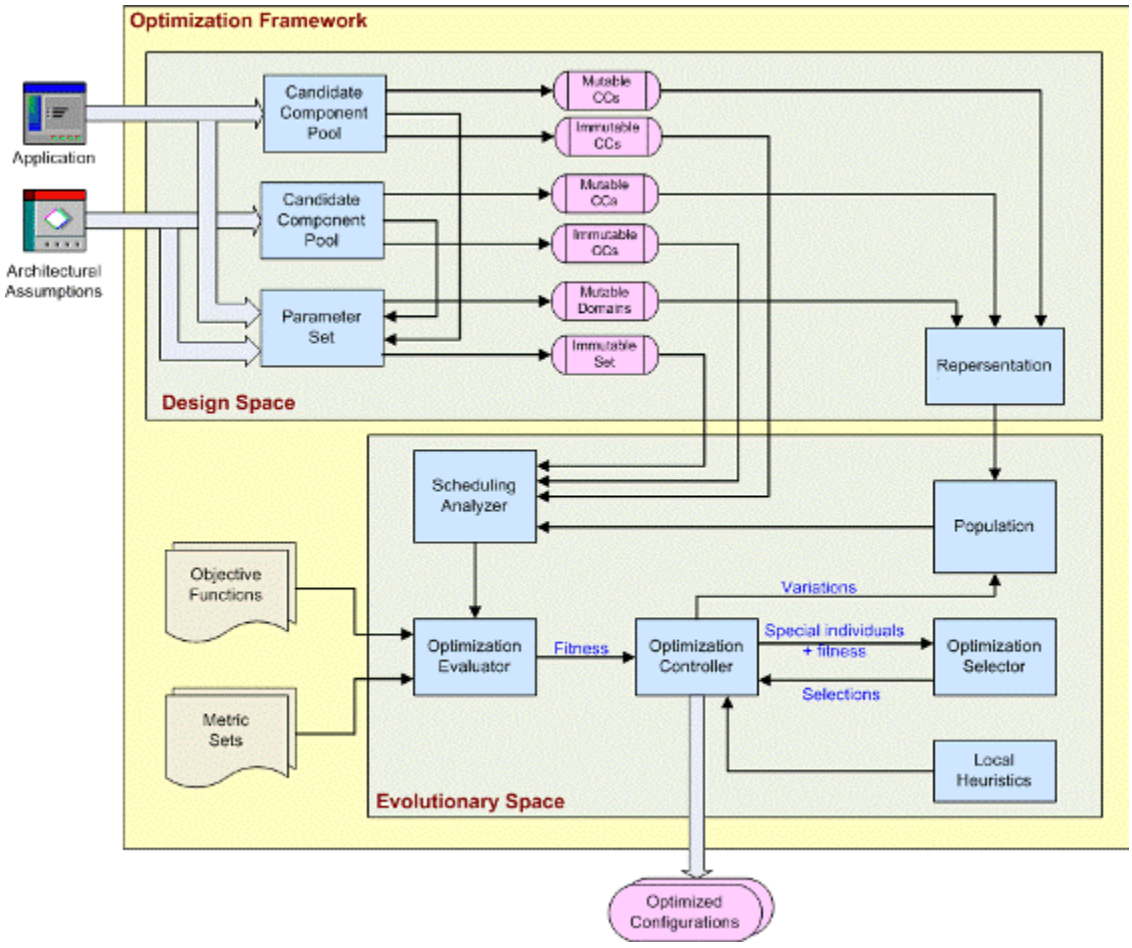


Figure 1: Preliminary optimization framework diagram.

2.4 Line Crossing Application

To illustrate the modeling and evaluation capabilities of our optimization framework as well as the implementation of derived WSN configurations on our prototype platform, we give an example of a WSN-based line crossing application.

2.4.1 Application Specification

The goal here is to have a system that can detect a moving object in the proximity of the sensors. In the line crossing application, all the sensor nodes should be placed in a linear network topology, so that it is also possible to determine when the moving object crosses that line.

2.4.2 Sensor Node Operation

Each node runs according to a TDMA schedule with the three operation modes of transmission, reception, and idle status. The sensor node enables its associated microphone sensor when it enters transmission mode and senses an acoustic signal from the environment. The corresponding digital data is then sent to a microcontroller through an A/D converter. A user-defined threshold is used to filter the sensed data. Whenever the sensed data is above the threshold, the data will be encoded with a data token from memory for transmission to the next node. More specifically, the data token will be encoded with a header as well as a node ID to form a complete message for transmission. The data token involved here is formatted with a separate bit for each node. Hence, once a token arrives at the base station, it can easily be decoded to determine whether a moving body has crossed the line of sensor nodes, and if so, which nodes were closest to the line crossing event.

In the reception mode, a sensor node needs to enable its transceiver and wait for a message to arrive from the previous node in the sensor array. Upon receiving a message, the node will decode it and compare its contents with the expected header and node ID. If both comparisons match, the data token will be stored into memory, and the node will wait for its next transmission slot according to the TDMA schedule.

When a sensor node enters the idle mode, the microcontroller powers down the transceiver and sensor devices, and then it enters a low-power mode to save power throughout the rest of the idle interval.

Figure 2 illustrates the operation of this WSN-based line crossing application, as well as the associated TDMA operations.

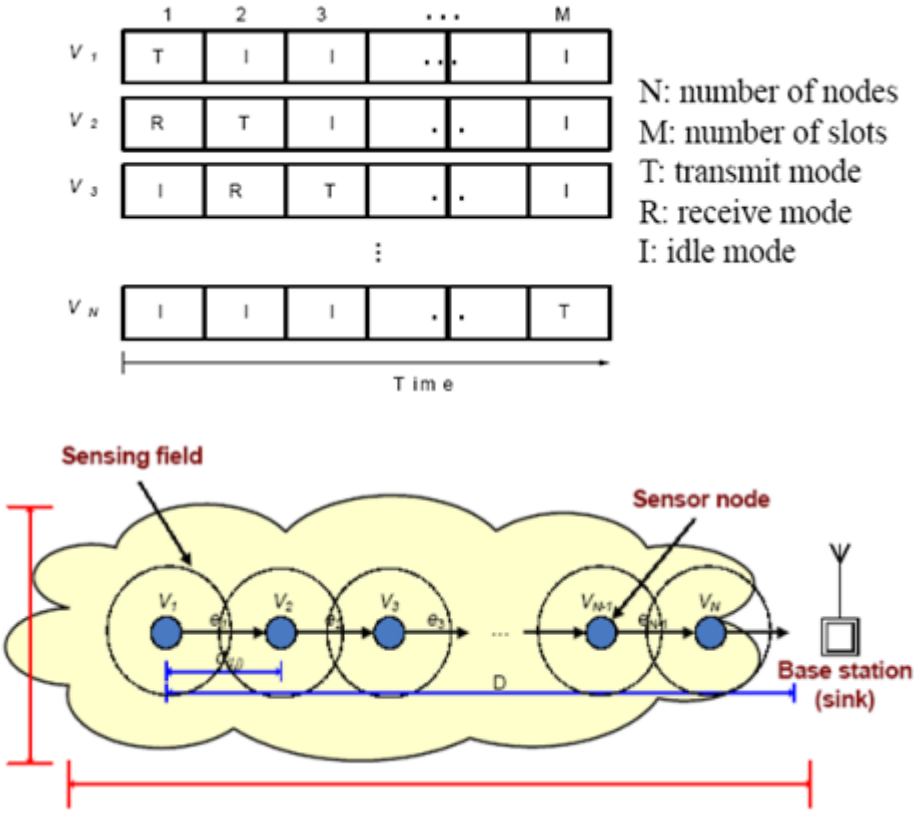


Figure 2: A WSN-based line crossing application with associated TDMA operations.

2.5 Optimization Framework Implementation

The optimization framework is built around the particle swarm optimization technique, which is extended to meet the design requirements of the line crossing sensor network application. Implementation aspects of the generic and application-specific features are described in this section.

2.5.1 Particle Swarm Optimization

Particle swarm optimization (PSO) algorithm was introduced by Eberhart and Kennedy [15] in 1995 as a population-based optimization technique simulating social behavior of individuals. The algorithm explores a problem space with a set of individuals, which are referred to as *particles*. The group of particles that are operated on is called the *swarm*. Particles in the swarm are "flown" through the D-dimensional search space to find an optimized solution. Each particle in the swarm constitutes a potential solution, and its multi-coordinate position, initialized to a random value, is given at any iteration of the algorithm by $X_i = (x_{i1}, x_{i2}, x_{i3},..., x_{iD})$, where i is the particle's index. A particle's best position achieved so far is also recorded as $P_i = (p_{i1}, p_{i2}, p_{i3},..., p_{iD})$, as well as the dynamic velocity $V_i = (v_{i1}, v_{i2}, v_{i3},..., v_{iD})$, with which the particle moves in the search space, influenced by the swarm's global best solution and the particle's own best solution. At each iteration, the particle's position is evaluated with a fitness function that corresponds to the optimization objective rule of the problem. The index of the particle that achieves the best fitness value in the swarm becomes the global best g, and the swarm keeps a record of the best fitness value thus achieved.

The initial algorithm is studied and developed in [4][5[14][21], introducing some variations, mainly an inertia weight ω as a factor controlling the impact of the previous velocity on the current, and a local best version comparing only neighboring particles as an alternative to the global best.

2.5.2 Generic PSO Package

The PSO algorithm adopted in this work is given in Figure 3. $\varphi 1$ is the weight attributed to the cognition component and $\varphi 2$ is the weight attributed to the social component in the particle's velocity update equation. In our exploration tool, the algorithm is implemented as a generic optimization package using Java. In this package, the swarm of particles is initialized with random particle positions and velocities, and then iteratively operated on to explore the underlying design space until an exit condition is satisfied. Specifically, the condition for exiting is that either a solution is found having a cost function value that is within a pre-specified range, or a fixed maximum number of iterations have been completed.

The particles have multi-dimensional coordinates, where the position, velocity, and best position achieved so far are recorded for the particle on each of its dimensions. The position values that a particle can take on at each coordinate are explicitly bound by user-specified minimum and a maximum values. If during the optimization process, the particle position attempts to take on a value that is outside this range, the value is clamped to the corresponding limit. The same principle applies for the values of the particle velocity on each coordinate.

An interfacing class is included to serve as a connection layer between an application that uses PSO and the PSO package itself. The role of this interfacing class

```
PSO()
  Initialize swarm of particles with random positions and velocities
  Repeat
    For each particle i
       Evaluate fitness (Xi)
       if fitness(Xi) > fitness(Pi)
             Pi = Xi
             For each coordinate d
                  vid = ω.vid + rand().φ1(pid - xid) + rand().φ2(pgd - xid)
                  xid = xid + vid
    Compare particles fitness update g
    Global best fitness = fitness(Pg)
  Until stopping condition is met
```

Figure 3: Pseudocode of generic PSO-based search.

is mainly to convert input information from the application-specific format into swarm particle data, and to format swarm results to be displayed as output. The general interaction among the swarm, the application, the inputs, and the outputs is shown in Figure 4.

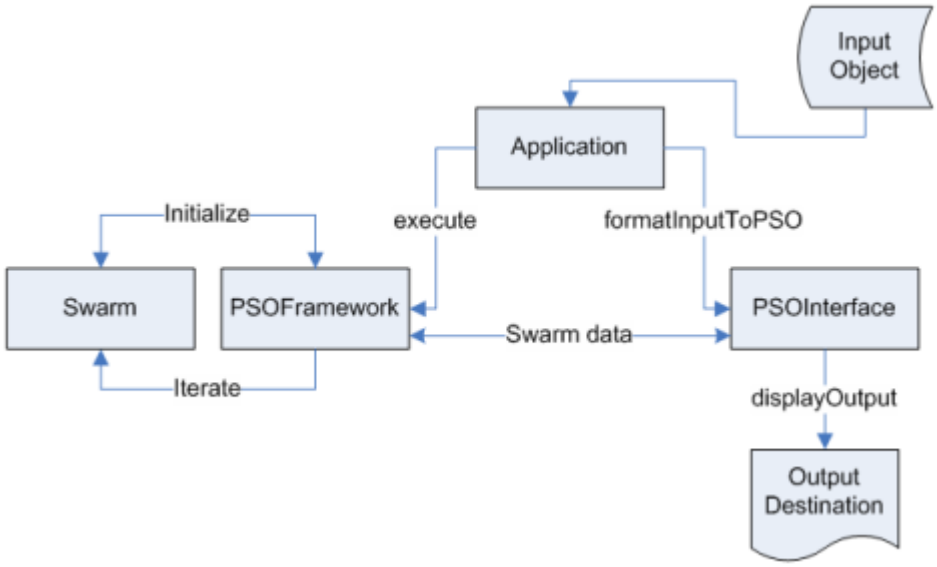


Figure 4: Flow of information in the PSO package implementation.

The application-specific implementation part consists of the development of the fitness evaluator (i.e., the mapping of candidate solutions into values of the relevant cost function), and of the move algorithm for particles.

2.5.3 Application Fitness Evaluation

The fitness evaluator complies with the requirements of the line-crossing application model described in section 2.4, with the cost function being a scalar evaluation of energy models derived for the sensor network. In order for the framework to find an effective application-specific sensor network configuration based on energy consumption considerations, a node-level energy model is developed for data acquisition, computation, and packet transmission, as well as a network-level energy model that minimizes overall power consumption while maintaining application functionality. It is implied here that the power strength of the signal received at a sensor node still has to be greater than the specified threshold when evaluating transmission energy at the adjacent node, for any separating distance being evaluated. We also take into consideration time limitations for transitions between different node modes (transmission, reception, and idle), and for transitions between the microcontroller power modes when defining TDMA slot time. With all this in mind, we integrate various energy models from individual WSN layers, such as the physical, data-link, network, and application layers to consider a broad range of parameters that may affect system-level energy consumption. Alternative system configurations can then be evaluated by running simulations for estimating system-level energy consumption for a selected schedule of transmission, reception, and idle sequences. The resulting approach

to modeling helps us explore the design space of a sensor network application in a more comprehensive way.

The current implementation only considers a single objective, that of minimizing energy consumption, therefore local heuristics need not be applied when evaluating potential solutions' fitness.

2.5.4 Application Move Algorithm Implementation

The implementation of the move algorithm primarily updates the velocity and position of each particle on its respective coordinates according to the equations given in the algorithm of Figure 3, while constraining the velocity and position values to remain within maximum and minimum values provided from the application input data. An important feature added to the move algorithm is the handling of coordinates that have discrete position values. The property of a coordinate having continuous or discrete values is also retrieved from the formatted input. Discrete coordinates are designed as references to indexed structures where the actual discrete values are stored in increasing order. Particle positions on such coordinates in the swarm then represent the associated indices, and are updated by the move algorithm such that velocity increments or decrements occur in integer steps, moving to a higher or lower index. In the fitness evaluation process, the discrete value stored at the index is retrieved and used in the cost function.

2.5.5 WSN Application Representation in PSO

Mapping the application components to PSO elements requires other application-specific additions to the generic PSO model, mainly the extension of the

coordinate class model to include properties pertaining to the sensor network application, and the extension of the generic particle model to also reflect a WSN-related organization of the coordinates. In particular, a WSN particle's coordinates are a representation of the sensor node properties and of the network parameters being optimized.

Each WSN-related coordinate acquires a component name, such as RADIO, and a coordinate name, such as DATA_RATE or OUTPUT_POWER, which are two parameters chosen to be optimized for a node's radio component. That way, the complete set of relevant properties for a sensor node can be parsed from the application input data, and each property stored in a particle’s coordinate. An example of mapping the formatted input to the node representation as a set of PSO particle’s coordinates is given in Figure 5.

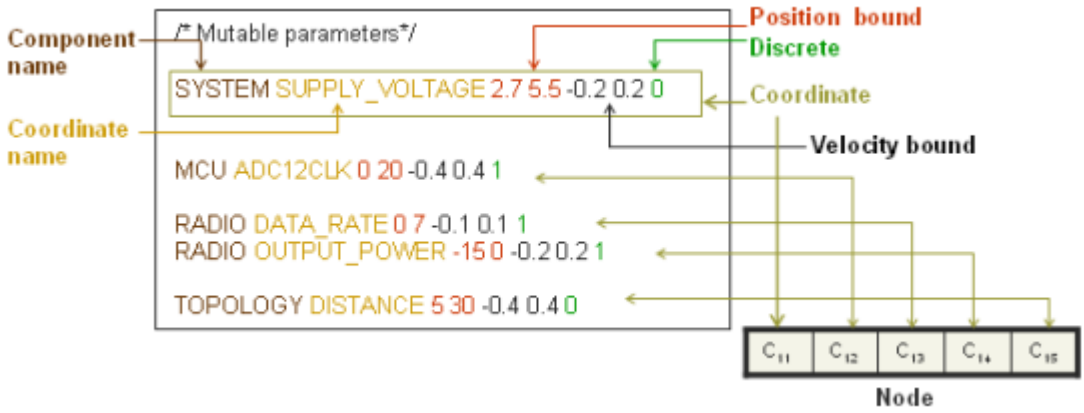


Figure 5: Mapping of input to PSO particle’s coordinates.

Then, this group of WSN-related coordinates is replicated as many times as the number of sensor nodes for a wireless sensor network application, and appended to the

WSN particle's coordinate array. Figure 6 illustrates the general representation of a WSN-related particle's format, which we use to store our optimization framework's mutable parameters. Here, we assume that we have *N* sensor nodes in the system, and *M* mutable parameters for each node.

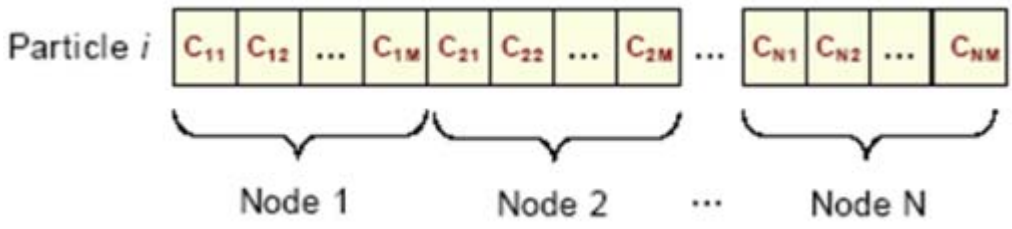


Figure 6: Representation of mutable configurations in WSN-related particles.

With such a way of representing WSN-related mutable parameters, sensor nodes in a WSN application can have numerous configurations, so that a detailed system-level optimization for the entire system can be carried out through the framework, and the resulting solution can be translated into a complete sensor network application by implementation on the targeted hardware platforms.

To retrieve information from coordinates in this model, the WSN particle offers methods to retrieve swarm data by providing a node's index to get a particular node configuration, or a coordinate name to compare configuration values associated with the same element in different network nodes.

The UML representation of our PSO implementation is offered in Appendix A.

2.5.6 PSO Testing and Operation

To verify the basic functionality of our PSO implementation, we used the Schaffer F6 benchmark:

$$f(x, y) = 0.5 + \frac{(\sin(\sqrt{x^2 + y^2}))^2 - 0.5}{(1 + 0.001(x^2 + y^2))^2}, -100 \le x, y \le 100$$

This is a function that is well known for its highly nonlinear nature, and is recognized as a fundamental benchmark for PSO techniques (e.g., see [7][22]). In our tests, the two-dimensional coordinate positions were bound by the values -100 and +100, while the velocity varied between -10 and +10. A solution could be found with less than 1000 iterations with a high success rate. Figure 7 illustrates the current position value versus iteration number for the Schaffer F6 benchmark from our PSO implementation, as well as the convergence behavior that we observed.

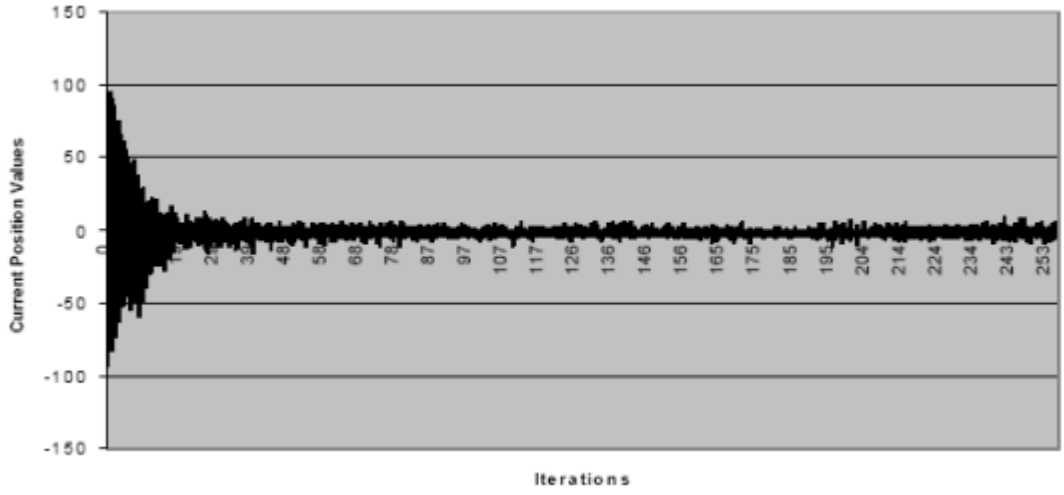


Figure 7: Convergence behavior of Schaffer's F6 benchmark function.

2.6 Optimization Framework Simulation Results

For evaluating WSN-related optimized configurations, we implemented the line-crossing application described in section 2.4, and we conducted experiments to compare energy consumption results associated with simulation of the PSO-based optimization framework, and measurements from a constructed prototype platform. For the prototype WSN testbed, we used an ultra-low power MSP430 microcontroller from Texas Instruments [26] and a transceiver from LINX Technologies [17]. In the simulation and in the measurement scenarios, we considered five critical parameters on each node: supply voltage, ADC clock frequency, data rate, output power, and transmission distance. These parameters were all treated as mutable for the purpose of optimization. In addition, we employed in our experiments a set of immutable parameter values, as listed in Table 1.

immutables	values	immutables	values
$N_{cpu-active\|tx}$	579	$N_{cpu-active\|rx}$	309
$t_{cpu-active\|tx}$	$0.072ms$	$t_{cpu-active\|rx}$	$0.039ms$
f_{clk}	$8MHz$	t_s	$125ms$
C	$100pF$	R	$9600bps$
$I_{cpu-sleep}$	$200uA$	$I_{radio-sleep}$	$200uA$
$I_{sensor-sleep}$	$200uA$	I_0	$2mA$
I_{ADC}	$300uA$	I_{UART}	$300uA$
I_{timer}	$300uA$	t_{sample}	$1.2us$
$t_{radio-startup\|tx}$	$3mA$	$t_{radio-startup\|rx}$	$6ms$
$t_{ADC-active}$	$53.6us$	$t_{radio-active}$	$6.33ms$

Table 1: Immutable parameter values.

We compared the results from the simulation of the optimization framework and measurement from the corresponding implementations on our WSN platform. For these comparisons, we chose 20 particles with $\varphi 1 = \varphi 2 = 2.0$ and $\omega = 0.95$ when running the PSO optimization algorithm for the experiment. The experimental energy consumption values for the optimized configurations for the 5-node line-crossing application through our optimization framework are shown in Figure 8, along with the simulation estimations.

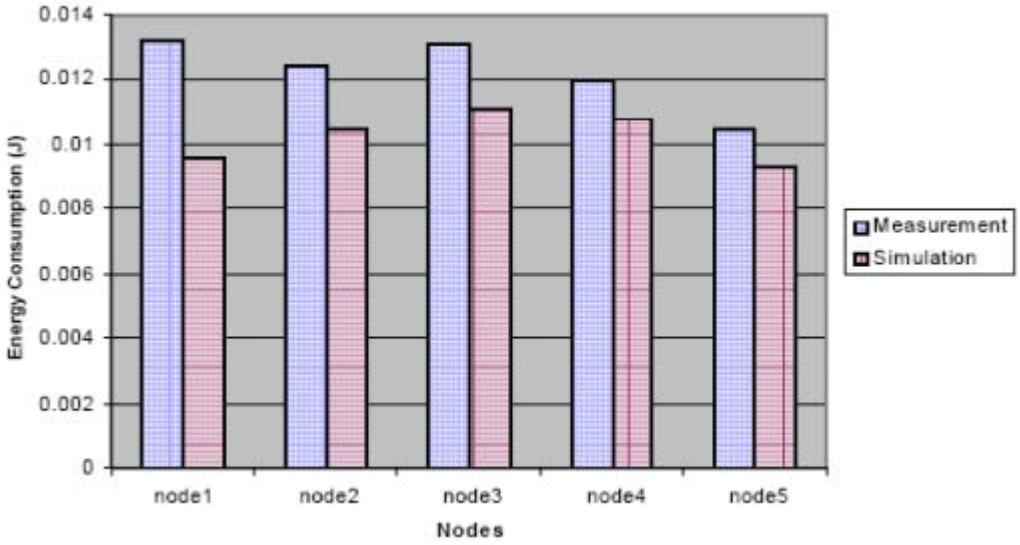


Figure 8: Energy consumption over the 5-node line crossing application with optimized configurations.

We conducted our test runs in the framework by varying the tightness of the binding restriction around the target "optimum" value. That is, since our fitness function measures the absolute offset from a pre-specified target value, we changed the range in which a fitness value that is not exactly equal to the target will nonetheless be considered as an acceptable solution, and trigger termination of the search.

We noted the number of iterations the program performed before reaching a solution within the range of each exit bound imposed on the search. For each binding constraint, 10 runs were performed, and the average number of iterations from these runs is shown in Table 2.

Also, the ratio of the number of solutions found out of those 10 runs was recorded when we conducted the tests for each of the binding constraints. Those values are listed as percentages in Table 3.

binding constraints	±0.0011	±0.00115	±0.00117	±0.0012	±0.00125	±0.0013
iterations	5.67	10.5	8.5	6.43	5.25	6.67

Table 2: Number of iterations to find the solution using various binding constraints.

binding constraints	±0.0011	±0.00115	±0.00117	±0.0012	±0.00125	±0.0013
Rate of succeed (%)	30	50	60	70	80	90

Table 3: Percentage of runs that found a solution using various binding constraints.

According to the results in tables above, we observe that the algorithm either converges at a very early stage or does not find a solution at all in 4000 iterations. This observation shows that PSO can explore the design space with a high convergent speed if the search engine does not fall into a local minima stage. To prevent our optimization framework from falling into any local minima stage is part of our future research work. However, the results for the rate of success confirm our expectation that a tighter binding constraints around the target is achieved with a smaller percentage of successful runs.

Chapter 3: Dataflow Context Switching

This chapter presents a preliminary effort in integrating synchronous dataflow concepts with multi-tasking support across multiple graphs, by the development of a context switching model that handles scheduling and controls concurrent execution of multiple application graphs.

3.1 Dataflow Graphs in Digital Signal Processing

Since dataflow graphs are a main component in the model we present, general concepts related to dataflow graphs and their utility in digital signal processing (DSP) are briefly reviewed here.

3.1.1 Dataflow Graphs

A dataflow graph is a directed multigraph [24]. Vertices in the graph, called actors, represent computations, while edges, also called arcs, represent FIFO-queued data values directed from one computation's output to the input of another. This representation captures data precedence between computations. The data values, usually referred to as tokens, arrive at an actor's input and are consumed by that actor, which performs computations on them. This action is called *firing* of the actor. A certain number of tokens is then produced on the actor's output.

3.1.2 Utility of Dataflow Graphs for DSP

Embedded applications in DSP face more limitations than general purpose computation, thus emphasizing the need to exploit special application characteristics in order to optimize the design and implementation for the specific set of constraints that

must be satisfied [24]. Specialized computational models such as dataflow variants expose relevant application structures and provide an intuitive specification mechanism for DSP. Dataflow also has the potential to enable effective parallelism detection by the compiler, and to simplify the static scheduling techniques because of compile time predictability.

In particular, when the application has no decision making at the task level, it can be represented by a synchronous dataflow (SDF) graph. SDF is a special case of the general dataflow model. With the SDF computational model, it is possible to check deadlock conditions and to determine whether the implementation is possible using a finite amount of memory, in exchange for the limited expressivity. Scheduling is also simplified, as the sequence of actors firing can be determined statically (at compile time) such that all precedence constraints are met and all the arcs' buffers return to their initial states. A set of algorithms that compile dataflow programs for embedded DSP applications into efficient implementations, focusing on minimization of code size and buffers memory, is given in [3].

3.2 Context Switching Model

Given that dataflow graphs have proven to be useful for DSP specification, and that sensor network applications use in many instances a substantial amount of signal processing, we draw on this fact to expand the usefulness of dataflow by using operating systems capabilities.

3.2.1 Motivation

Motivated by the potential effectiveness of a dataflow programming model for implementing sensor network applications and by the fact that only one instance of dataflow program can run on an embedded processor at a time, we propose the context switching model for concurrent graph execution. The model's goal is to allow multiple applications to be executed in an intermittent way by sharing resources, instead of running them sequentially, which may violate real-time deadlines.

3.2.2 Model Theory

The dataflow context switch model is designed to handle concurrent runs of multiple dataflow graphs and events servicing on a single embedded processor. By collecting individual graph priorities, scheduling information, and resource usage at compile time, the model allows a context switch operation to occur only at specific switching points inserted statically. It is also at those switching points that polling for selected events takes place, followed by a controlled servicing of those events.

3.3 Proposed Scheduling Operation

A key point in the model is to allow interrupting execution of an application graph in order to start or resume execution of another one, and also service pending events, all within an ordered structure defined at compile time. Consequently, we need to determine an acceptable period to regulate context switching operations, and also a global scheduler to overlook the execution sequence of graphs and event service routines.

3.3.1 Switching Points

Switching points are locations in the graph schedule where a possible switch to another graph execution and events servicing can be performed. Those switching points are separated by a maximum time interval of T, the predefined switching period. T is chosen such that the frequency of context switching operations does not cause a latency overhead that violates time deadlines of individual graphs. Also, T cannot be larger than the maximum time that pending graphs or events can wait for before acquiring the processor. Once T is determined, each application graph schedule is then traversed, and a switching point is inserted statically before each actor or group of actors whose total execution time is less than T. In the case where a single actor's execution time is greater than the switch period, a switching point is inserted, and the actor is also preceded by a timer that will set off asynchronously once a time equal to T has elapsed. The timer is made to reset dynamically as long as the remaining actor time is greater than the switch period. Whenever a timer sets off at runtime, the graph execution schedule is interrupted and polling for events takes place. Figure 9 shows a graph with inserted switching points and timers.

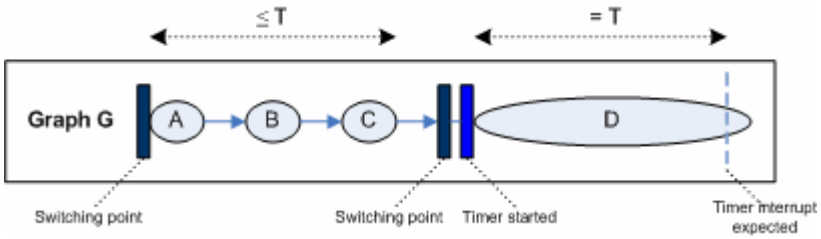


Figure 9: Switching points and timers insertion.

3.3.2 Events and Interrupts

Graph-related events or independent interrupts can arise while the application is running. Some interrupts require immediate attention of the processor and have to be serviced asynchronously. On the other hand, most application-generated events and some other interrupts can be more or less delayed, depending on how critically they affect the course of execution. In our model, we consider those events and interrupts that can be delayed, and classify them according to priority levels. The highest priority is attributed to the most urgent ones, whereas less critical events and interrupts are assigned a lower priority. A smaller priority number depicts a higher priority level. When events or interrupts occur during execution, they are stored in a FIFO queue pertaining to their priority level. They will be serviced once a switching point is reached or a timer expires in the execution schedule. However, in order to prevent other tasks from waiting for an unaffordable time while events are being serviced, we restrict the number of events and interrupts handled at any particular switching point to N_i, for priority level i, and choose $N_i > N_j$ for $i < j$; and we also assume that nesting of events is not supported.

3.3.3 Application Graphs

Depending on specific factors, such as earliest deadline for example, graphs are assigned one of several priority levels, while maintaining that graphs priority levels are lower than those of events and interrupts. As for events and interrupts, FIFO queues handle precedence order between graphs of equal priority levels and execution starts with the queue having the highest priority. Each actor in the executing graph schedule

sequence runs to completion as long as there is no timer scheduled to interrupt it. When a switching point is reached in the schedule, that graph may be suspended by the global scheduler. Therefore, a graph can be in one of the states presented in Figure 10. The state of the graph can change dynamically. After a switching point, the interrupted graph can resume execution, or another graph can be started, if any are in the ready state. On return from a timer interruption, the suspended graph resumes execution of its current actor.

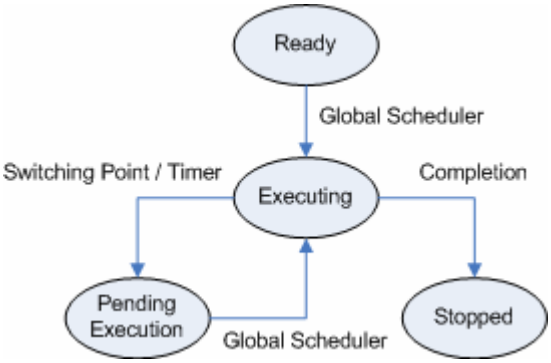


Figure 10: Possible states of a graph.

3.3.4 Global Scheduler

Control is relayed to the global scheduler at a switching point, or at a timer interruption if it is determined that events have taken place during graph execution. The global scheduler checks the priority 0 events list first, and runs the corresponding events service routines when needed, up to N_0 of them. Then events from the priority 1 events list are serviced, up to N_1 of them, and so on. When events handling is complete, if an actor has been interrupted by a timer, then this actor resumes execution. Otherwise, the global scheduler selects a graph either from the list of graphs pending execution, or

from the list of graphs ready for execution at the same or higher priority level. The list of graphs is determined at compile time and they are dispatched starting with the highest priority ones. Therefore, graphs of higher priority level can only be present in the ready list if the dynamic increment of graph priority is supported.

The information that the global scheduler needs at compile time is the identification of application graphs, their priority levels, and the location of their schedule execution code, in addition to the events supported, their respective priority levels, and the routines that service each of them. An illustration of a global scheduler operation is shown in Figure 11.

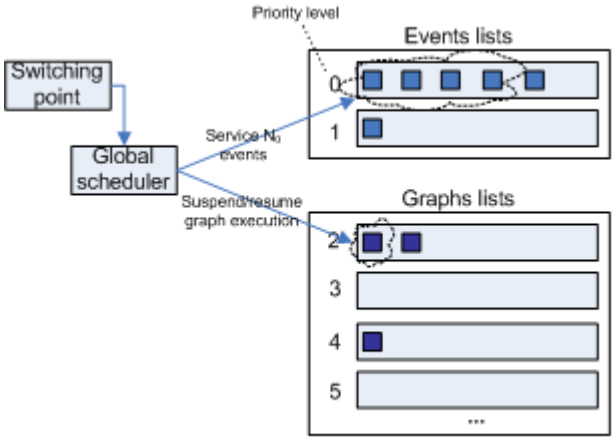


Figure 11: Global scheduler operation.

3.4 Algorithms

This section presents the algorithms developed for the dataflow context switching theory presented above. Those algorithms are adopted in our current implementation of the model.

3.4.1 Insertion of Switching Points

The considerations for inserting switching points in a graph schedule have been detailed in section 3.2.1. We list in Figure 12 a formal algorithm that, given a dataflow graph schedule and a switching period T, determines before which actors of that graph a context switching operation can take place. Each actor is assumed to have an execution time attribute related to the computation it performs. The algorithm also specifies that a timer is to be started when the actor execution time is greater than T. The timer is restarted dynamically during the execution of long actors if the remaining actor time when it sets off is still greater than T.

```
InsertSwitchingPoints(Schedule S, SwitchingPeriod T)

  CurrentActor = S.getNextActor()

  While S.hasMoreActors do
         TotalTime = getExecutionTime(CurrentActor)
         Insert switching point before CurrentActor

         If TotalTime > T
                   Insert timer before CurrentActor
                   CurrentActor = S.getNextActor()
         Else
                   While TotalTime < T and S.hasMoreActors do
                            TotalTime = TotalTime + getExecutionTime(S.getNextActor())
                            CurrentActor = S.getNextActor()

  Insert switching point
```

Figure 12: Switching points and timers insertion algorithm.

3.4.2 Static Graphs Interlacing

While the occurrence of events and interrupts is more likely to be determined at runtime, deciding whether to resume the interrupted actor or to start execution of another graph at a switching point can be achieved at compile time. This simplifies the

role of the global scheduler at runtime since the sequence of actors' execution is known statically, so it is only necessary to handle events and interrupts as described in section 3.2.2. When an alarm expires, control is also transferred to the event handling procedure. The algorithm executed at switching points for statically defined graphs transition is given in Figure 13.

```
SwitchingPoint()

  If events have occurred
        Call eventHandler()
  Execute next actor in schedule
```

Figure 13: Switching point operation algorithm.

3.4.3 Dynamic Graphs Interlacing

As an alternative to the static graphs interlacing approach, the global scheduler can take control of dynamically selecting which graph can be scheduled to run next, at the expense of somewhat more complex code. This scheme is particularly useful if graph priorities or other considerations change at runtime, favoring the execution of one actor over another. The global scheduler in this case needs to keep track of whether the graph is ready, pending execution, or stopped, in addition to accounting for the variable properties such as priority levels. The algorithm executed by the controller called at switching points for the dynamically scheduled graphs transition approach is given in Figure 14.

3.5 Implementation

```
Controller()

  If events have occurred
         Call eventHandler()

  If non-stopped graphs of higher priority exist
         Schedule graph G from highest priority graph queue
         Execute next actor in G
  Else
         If random() == 0
                  Resume execution of interrupted actor
         Else if random() == 1
                  Schedule graph G from same priority graph queue
                  Execute next actor in G
```

Figure 14: Global scheduler control algorithm.

A simplified implementation of the static and dynamic scheduling models is conducted to evaluate the operational applicability of the theory.

3.5.1 General Considerations

Our implementation utilizes the DIF and DIF-to-C packages capabilities for dataflow graphs specification, application scheduling, and automated C-code generation. The procedure for producing the overall code for each approach is detailed below. As a general note, context switching code relies on C language libraries that deal with events and interrupts in a low-level subroutine of a program. The main factor is to save the stack environment of the current graph execution before switching to any other code, then to effectively restore this environment on returning from the switch.

3.5.2 DIF and DIF-to-C Packages

Dataflow Interchange Format (DIF) is a language designed to represent a wide variety of dataflow models for DSP systems [11]. Being a standard vendor-independent language that fully captures essential modeling information of DSP applications, DIF

allows a comprehensive representation of both functional semantics and component properties of mixed-grain dataflow graphs. The information captured by a DIF specification of an application can be automatically converted to graph instances by the DIF Front-end tool in the DIF package. The package, a Java-based software packages developed by the DSPCAD group at the University of Maryland, also provides implementations for dataflow-based analysis, scheduling and optimization algorithms. Another interesting feature of DIF is the capability of porting DSP applications across design tools with a high degree of automation.

DIF-to-C is a software synthesis framework integrated into the DIF package that automatically generates C-code programs implementing DIF specifications of dataflow graphs [10]. C functions associated with actor computations are compiled with the generated graph implementation, and user-defined scheduling and buffering strategies are incorporated in the framework as well, to finally obtain an executable targeting various possible embedded processing platforms. The design flow of the DIF-to-C framework is illustrated in Figure 15.

3.5.3 Static Scheduling Model

The static schedule implementation aims at defining at compile time the execution sequence of actors belonging to the different graphs under consideration. This implies that resuming an interrupted graph or starting execution of another graph is a statically-established decision. The only dynamic part is the servicing of events and interrupts that occur at runtime. Application graphs are specified in DIF, and the execution time of every actor is included as an attribute in DIF to be used by the context switching scheduler. The first operation is to generate actors' firing sequence for

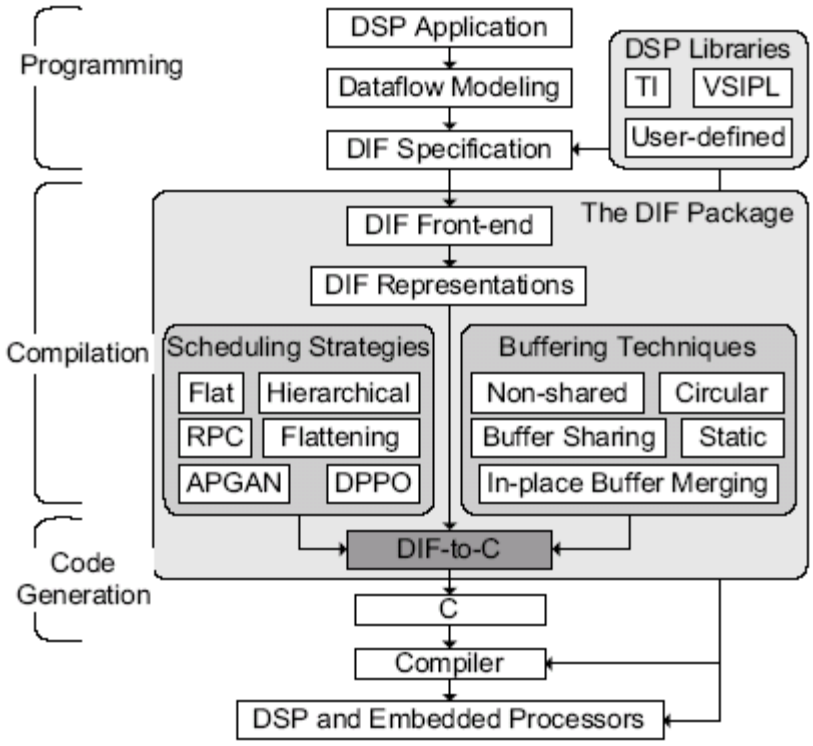


Figure 15: Design flow of the DIF-to-C framework.

individual graphs using the DIF package. Each firing sequence then serves as input to our java code that inserts switching points and timer locations between actors, as listed in the algorithm of Figure 12. Finally, all firing sequences are merged into one schedule where all the actors of all input graphs appear in addition to the switching points and timers determined to precede individual actors or actor groups. At the same priority level, graphs firing sequences are merged with the selection of the next actor determined randomly. On the other hand, merged firing sequences of actors at lower priority levels are appended after higher priority ones to ensure that the execution of those actors complies with the graphs priority rule introduced in section 3.2. The static scheduling model does not support any dynamic change of graph priority.

An example of obtaining the merged schedule for two dataflow graphs of same priority is given in Figure 16.

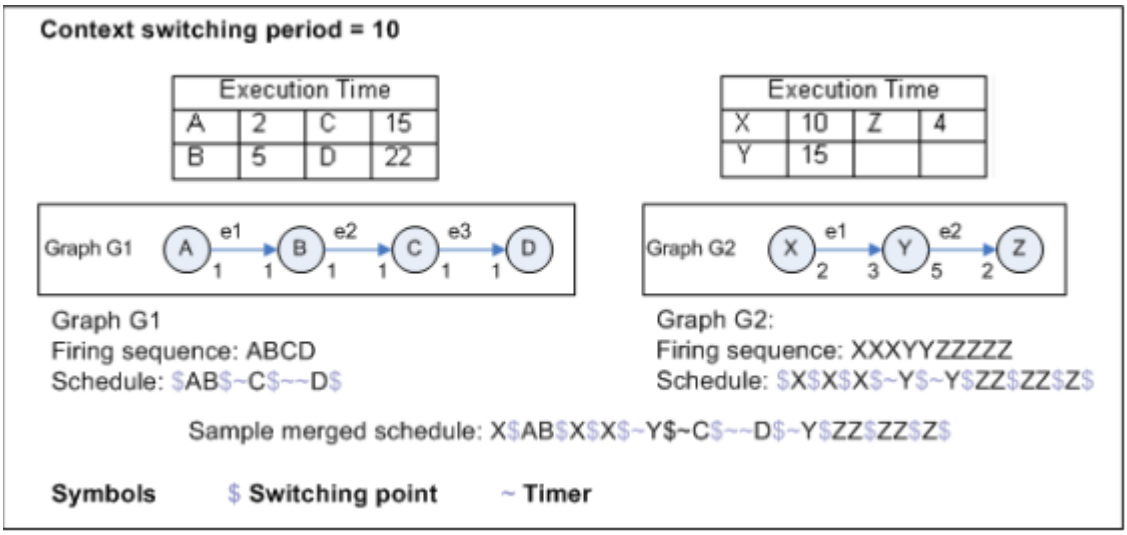


Figure 16: Example of statically merged schedule for two graphs G1 and G2.

The current version of the DIF-to-C package does not support automatic code generation for context switching. The work in this thesis represents a preliminary investigation that can form the basis for future incorporation of such code generation capability in DIF. In our experiments in this thesis, we use DIF-to-C to create the respective C-code for the dataflow graphs specified in DIF, which produces them as callable functions (not stand-alone applications), while the C-code for the computation associated with individual actors is assumed to be provided. At this point, C-code for graph transitions and event handling calls is inserted manually at the locations determined by the obtained static schedule. Additional C-code is also implemented for data structures, event support specifications, and a *main* function that initializes the global application and makes the call to start running the schedule. The overall C-code

functions are then linked and compiled into an executable file that can run on the embedded processor.

Figure 17 illustrates the general static scheduling process, given two input graphs G1 and G2.

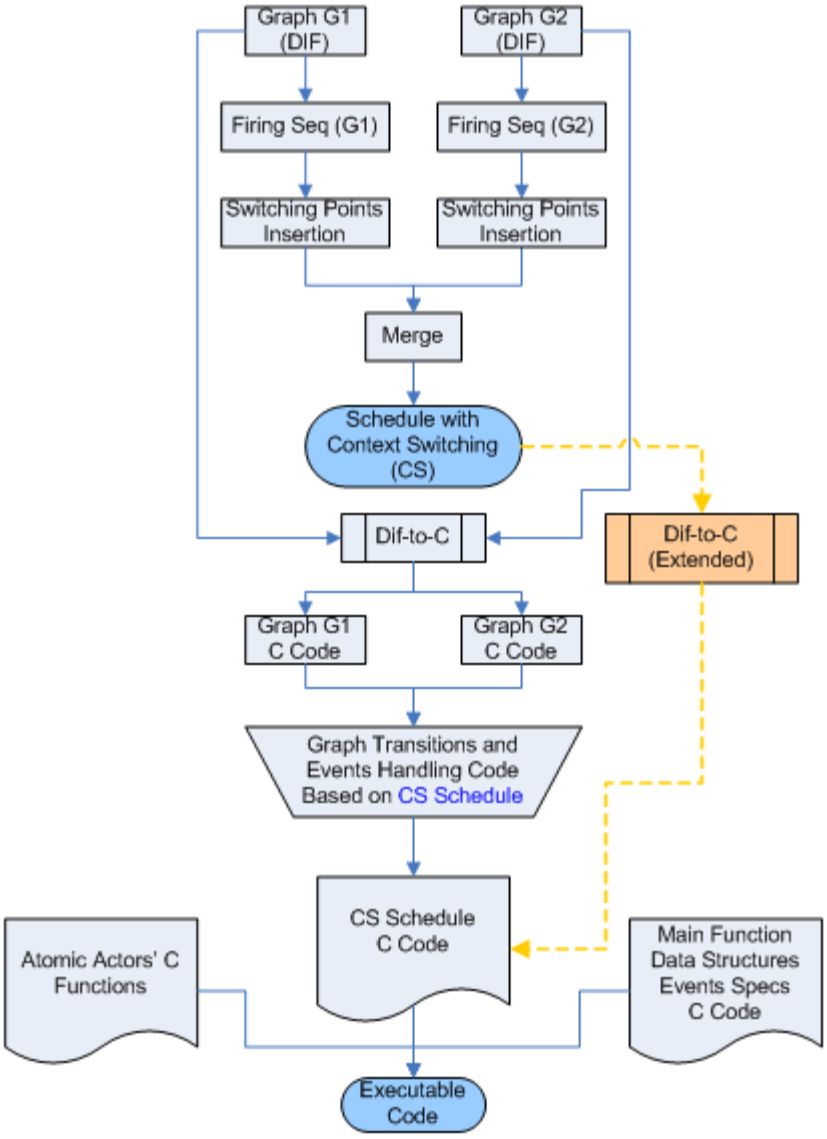


Figure 17: Static scheduling implementation process.

3.5.4 Dynamic Scheduling Model

In the dynamic scheduling implementation, we leave the selection of the next actor to be executed at a switching point to be determined dynamically by the global scheduler. In this scheme, the global scheduler is mainly a controller that is aware of any runtime modification to graph priorities or other considerations. Thus it can adapt the transitions among the different graphs to the dynamically changing application requirements or available resources, at the expense of an increased complexity overhead in the global scheduler code. A static schedule merging all graphs actors is not needed; however, defining switching points and timers insertion in each firing sequence is still required and carried out according to the algorithm in Figure 12. At any such interruption, a simple call to the global scheduler is placed manually in the DIF-to-C-generated graph implementations, since the automatic code generation for context switching elements has not yet been added to the package capabilities. As in the static scheduling model, data structures, event handling specifications, and the *main* function are provided in separate implementations, and actor computations are also assumed to be provided as external C functions. The global scheduler is implemented following the algorithm in Figure 14. Similarly, the implementations are linked and compiled to create an executable file that can run on the embedded processor. The overall sequence of operations is depicted in Figure 18.

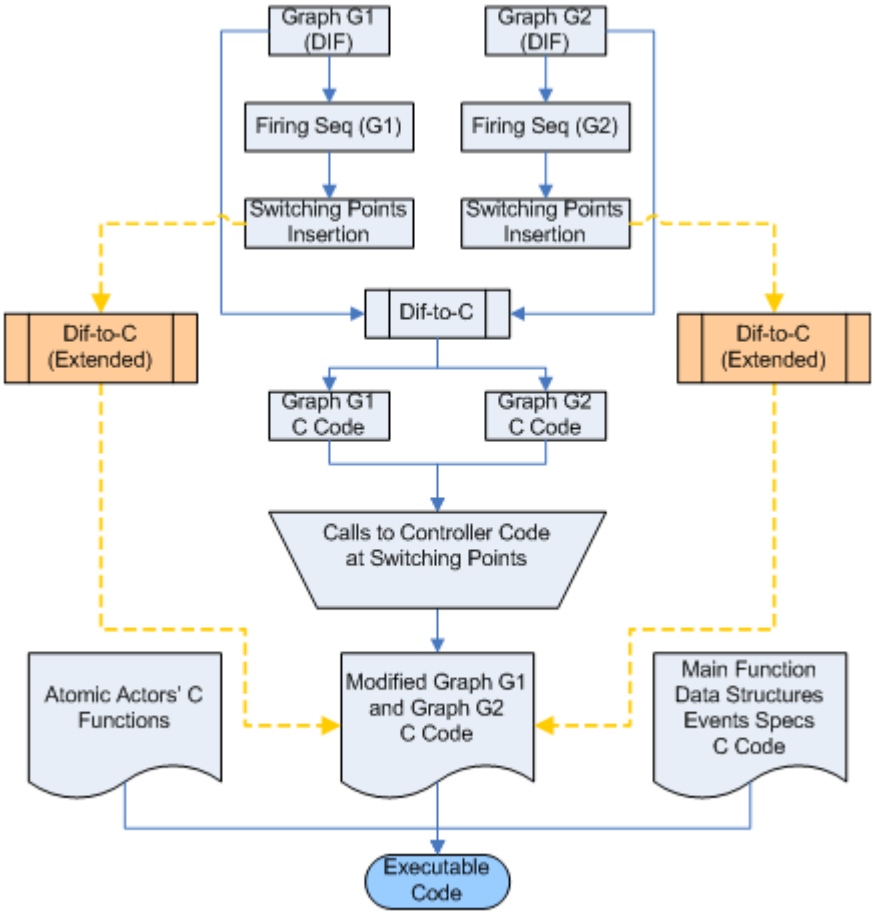


Figure 18: Dynamic scheduling implementation process.

3.6 Application Prototyping

The context switching model is tested with two DSP applications whose dataflow models are taken from Ptolemy II, a Java-based component assembly framework developed in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley [8]. The first application is the Spectrum application that gives a simple spectral estimation of the product of two

sinusoids in noise. The second application is the Maximum Entropy Spectrum that estimates three sinusoids in noise by three different techniques. DIF specification files are created for the Spectrum and Maximum Entropy Spectrum applications, respectively, which are then processed through our static and dynamic scheduling models implementation.

The illustration of the applications as provided in Ptolemy II is shown in Appendix B, and the complete DIF specification for each is listed in Appendix C. The output from the simulation of a statically-merged schedule of the two applications, with delayed servicing of software and hardware events, demonstrated the expected simulation sequence. Similarly, the simulation output of a dynamic schedule where the global scheduler switches between the two applications and handles events dynamically proved the model to be functional.

Chapter 4: Conclusion and Future Directions

We have presented in this thesis two models addressing the synthesis of embedded software for sensor nodes in specialized contexts. First, an optimization framework was designed to automate design-space exploration for wireless sensor networks' parametrized configurations. The framework was implemented based on the particle swarm optimization search technique and tested for the line crossing detection application. Results of this optimization framework (optimized network configuration) have been experimented with on a fully-functional sensor network prototype to verify the efficacy of the optimization methodology.

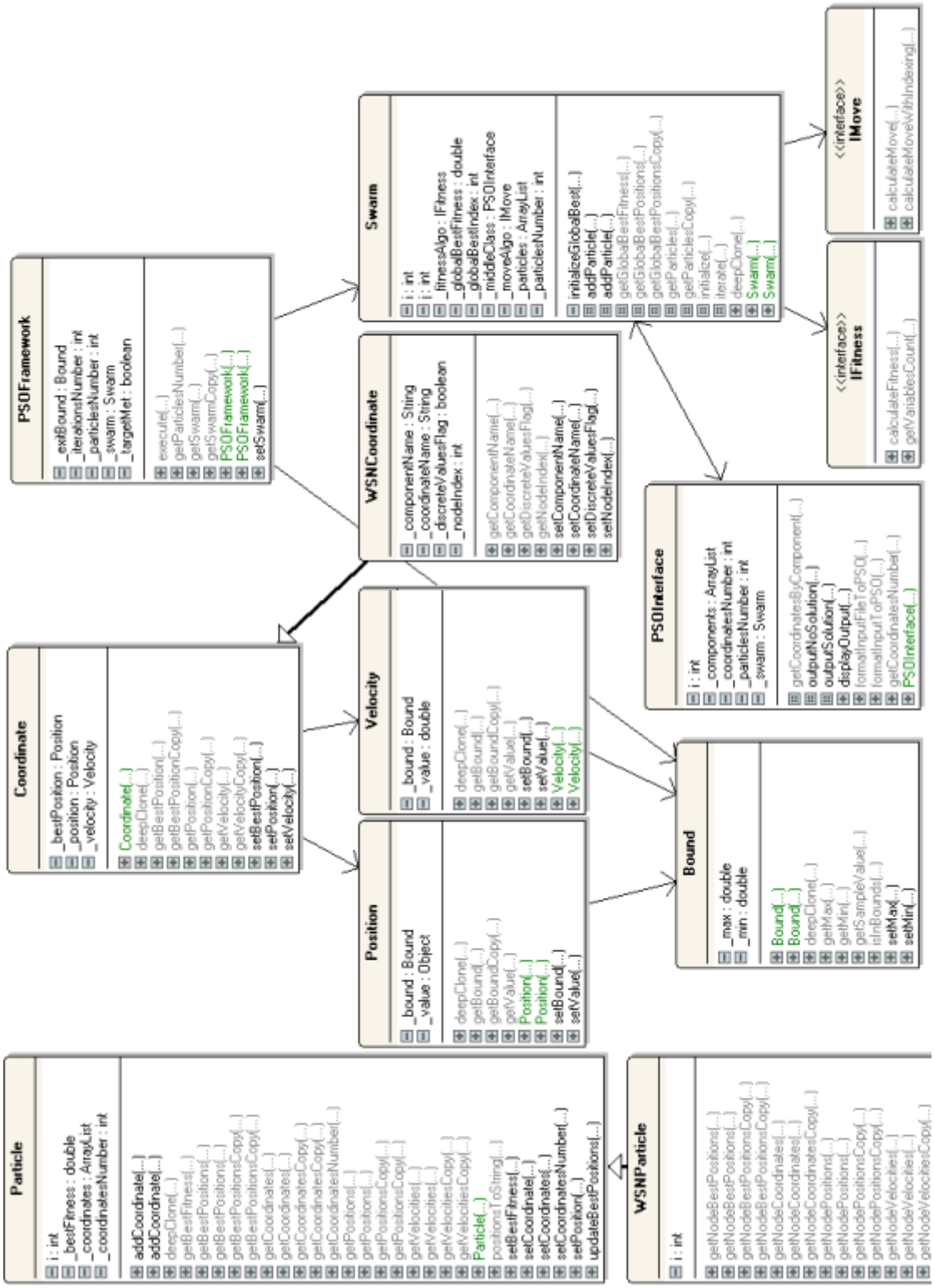
The second model was developed for a context switching mechanism that allows concurrent execution of multiple dataflow graphs on a single embedded processor. Static and dynamic scheduling approaches were implemented to test the functionality of the model.

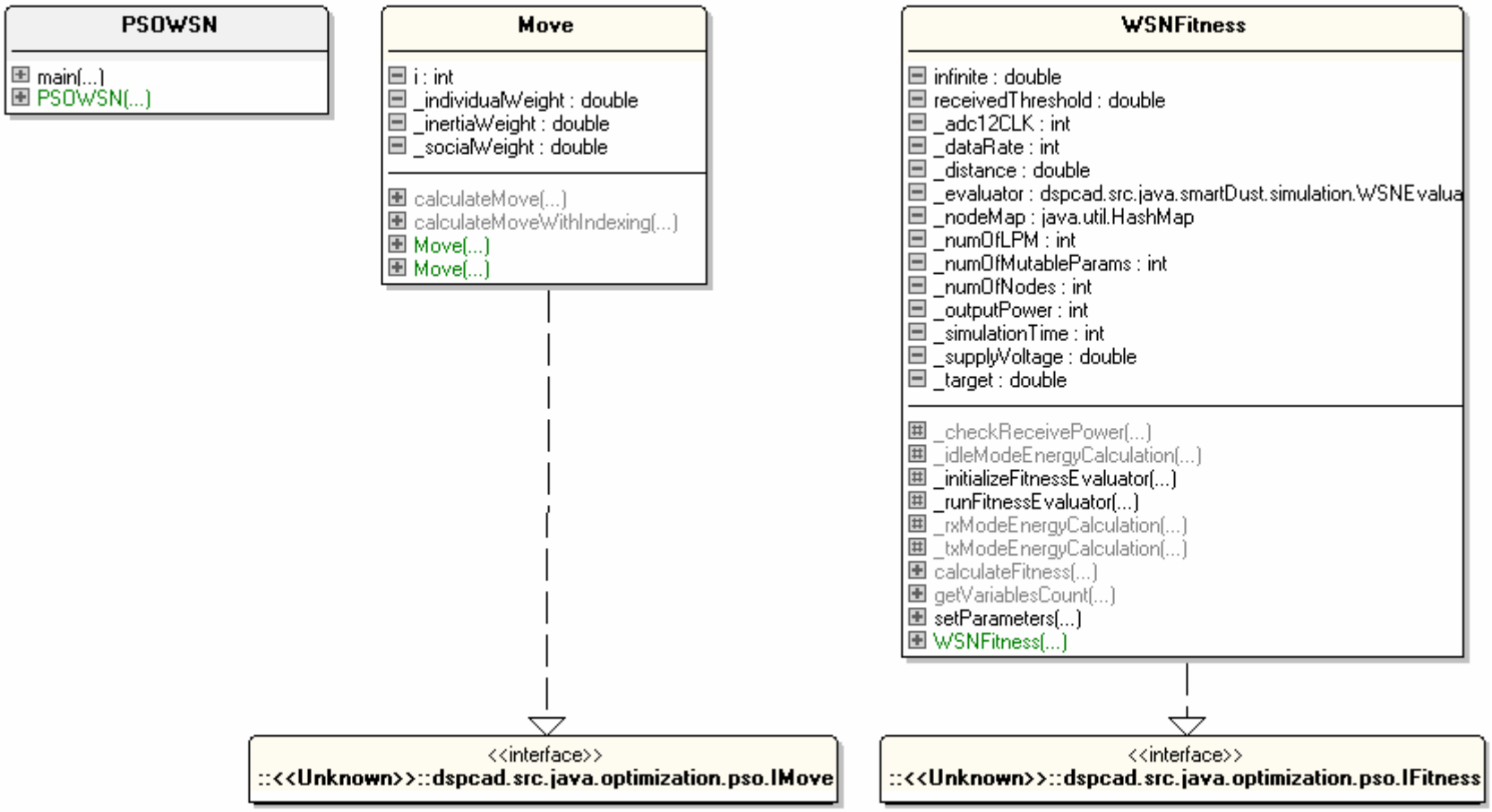
Simulation results in both synthesis contexts support the applicability of the adopted approaches for optimized operation of application-specific sensor nodes. In the PSO-based framework simulation runs, we were able to derive network configurations that met our optimization objective of minimizing energy consumption. When mapped to a hardware prototype implementing the line crossing application, those configurations yielded energy consumption measurements that matched the simulation results with a high degree of fidelity. Incorporation of additional system-level optimization metrics, such as cost, latency, and throughput, within our optimization framework constitute an important direction for further investigation. New models for such metrics can possibly be incorporated in conjunction with more sophisticated multi-

dimensional PSO strategies, so that our optimization framework can be more suitable to solve multi-objective optimization problems for application-specific wireless sensor network systems.

As for the context switching model that we introduced, it was demonstrated to be operational at the cost of a small amount of memory overhead. Not only did the model effectively use the information collected from the dataflow specifications at compile time to optimize runtime scheduling, but also it had enough flexibility to handle dynamically changing applications priority requirements. Additional effort can be invested to include dynamically-changing memory usage considerations that may impact the scheduling operation conducted by the global scheduler. Another useful feature would be to extend the DIF-to-C package in order to automate the code-generation of context switching C functions. Also, by considering additional test applications, the model can be further refined and evaluated for potential use with DSP applications on actual sensor nodes testbeds.

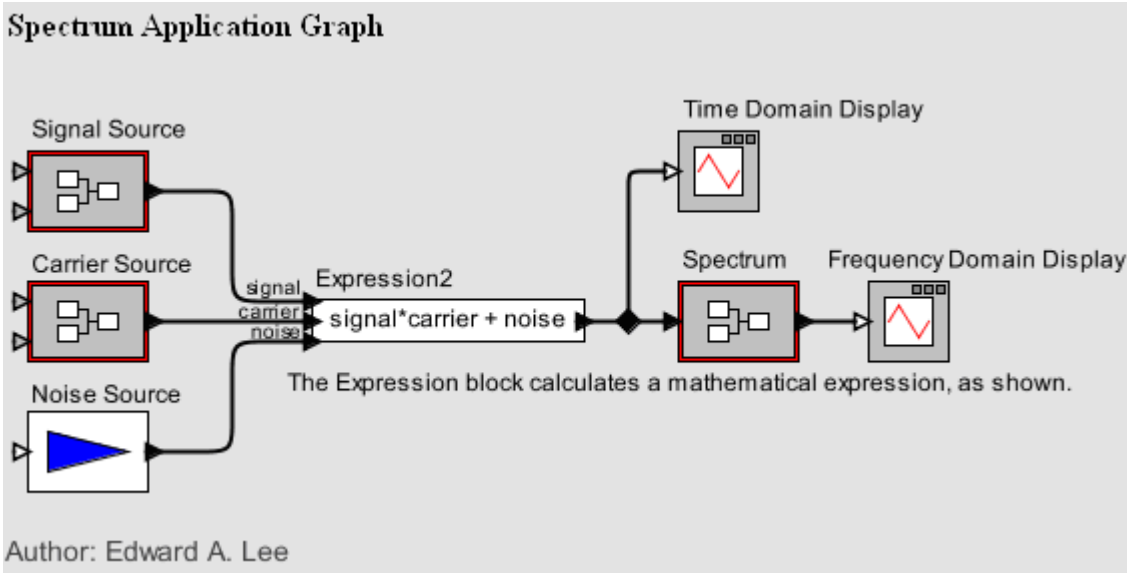
Appendix A: UML Diagrams of PSO Package

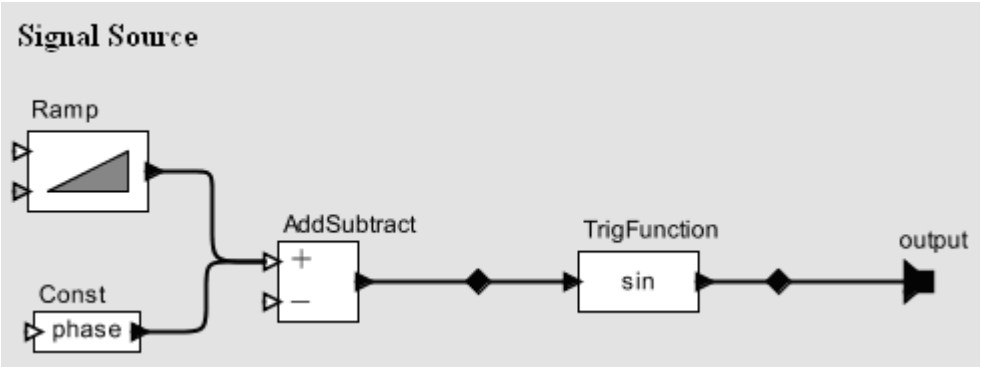


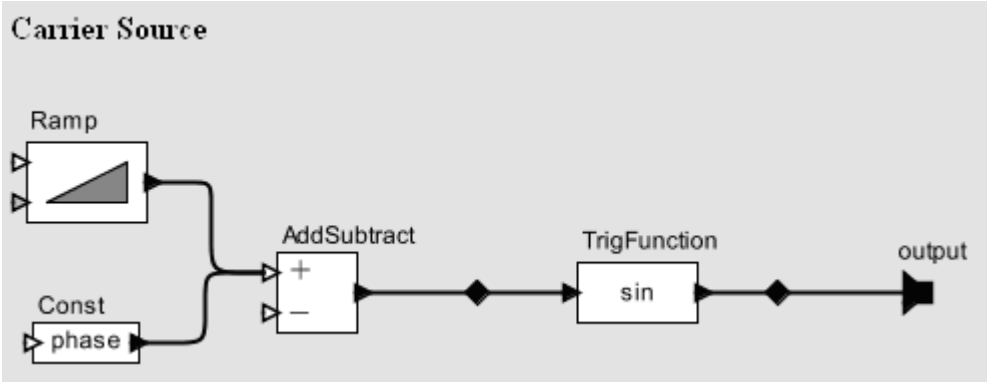

PSOWSN
main(...)
PSOWSN(...)
Move
i : int
_individualWeight : double
_inertiaWeight : double
_socialWeight : double
calculateMove(...)
calculateMoveWithIndexing(...)
Move(...)
Move(...)
WSNFitness
infinite : double
receivedThreshold : double
_adc12CLK : int
_dataRate : int
_distance : double
_evaluator : dspcad.src.java.smartDust.simulation.WSNEvalua
_nodeMap : java.util.HashMap
_numOfLPM : int
_numOfMutableParams : int
_numOfNodes : int
_outputPower : int
_simulationTime : int
_supplyVoltage : double
_target : double
_checkReceivePower(...)
_idleModeEnergyCalculation(...)
_initializeFitnessEvaluator(...)
_runFitnessEvaluator(...)
_rxModeEnergyCalculation(...)
_txModeEnergyCalculation(...)
calculateFitness(...)
getVariablesCount(...)
setParameters(...)
WSNFitness(...)
<<interface>>
::<<Unknown>>::dspcad.src.java.optimization.pso.IMove
<<interface>>
::<<Unknown>>::dspcad.src.java.optimization.pso.IFitness

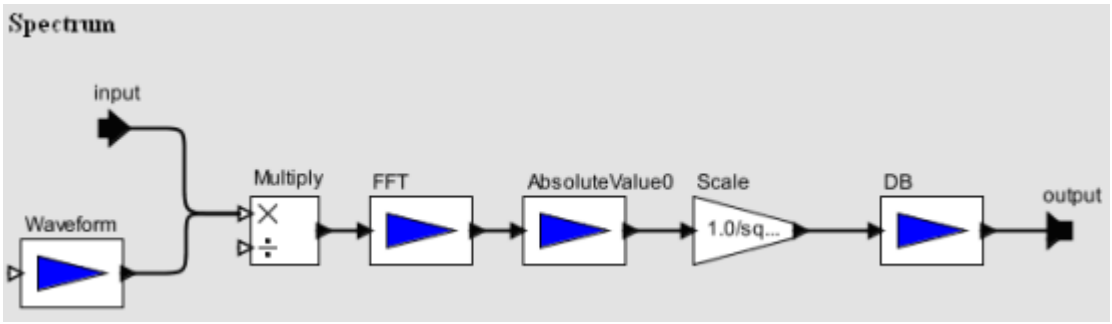
Appendix B: Spectrum and Maximum Entropy Spectrum Applications

B.1 Spectrum Application from Ptolemy II

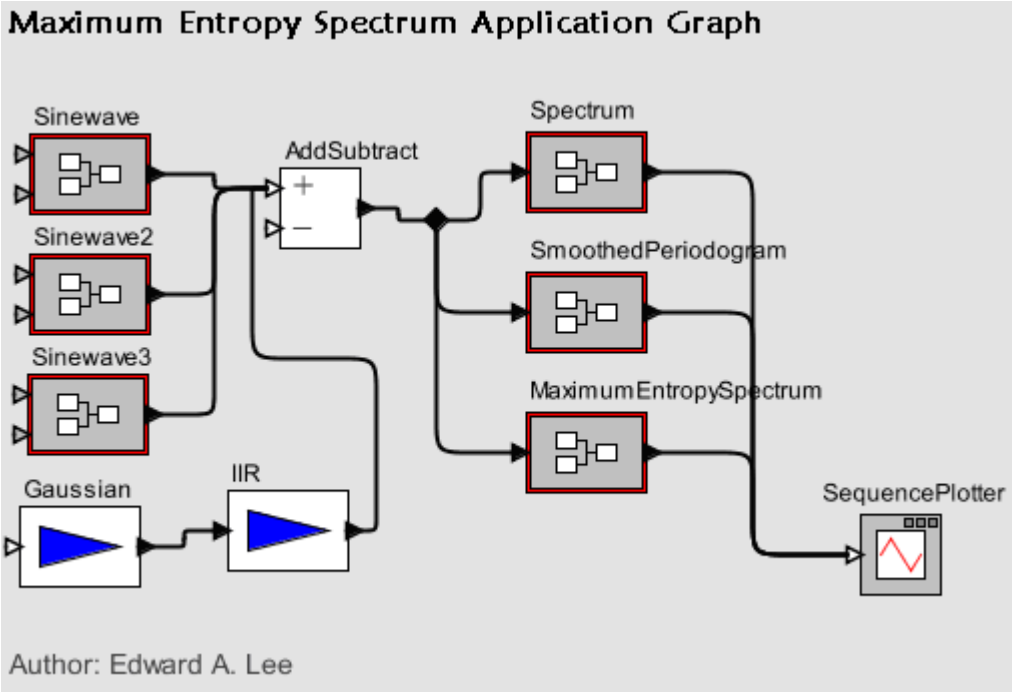


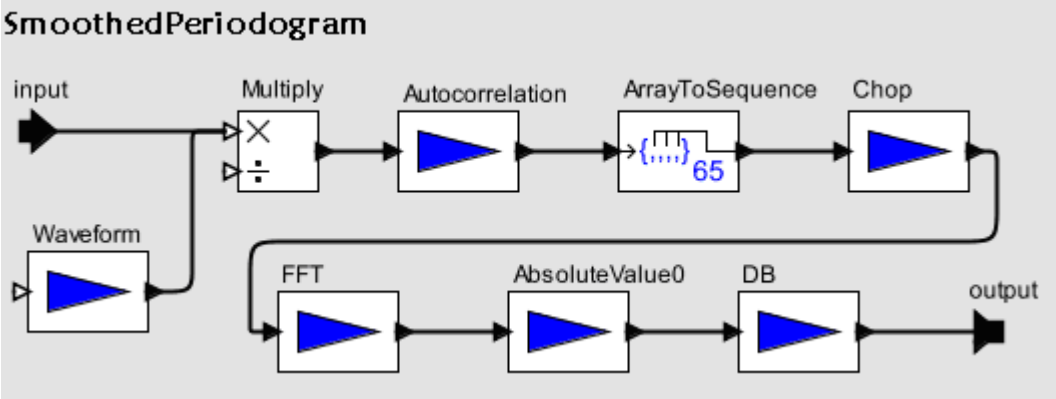


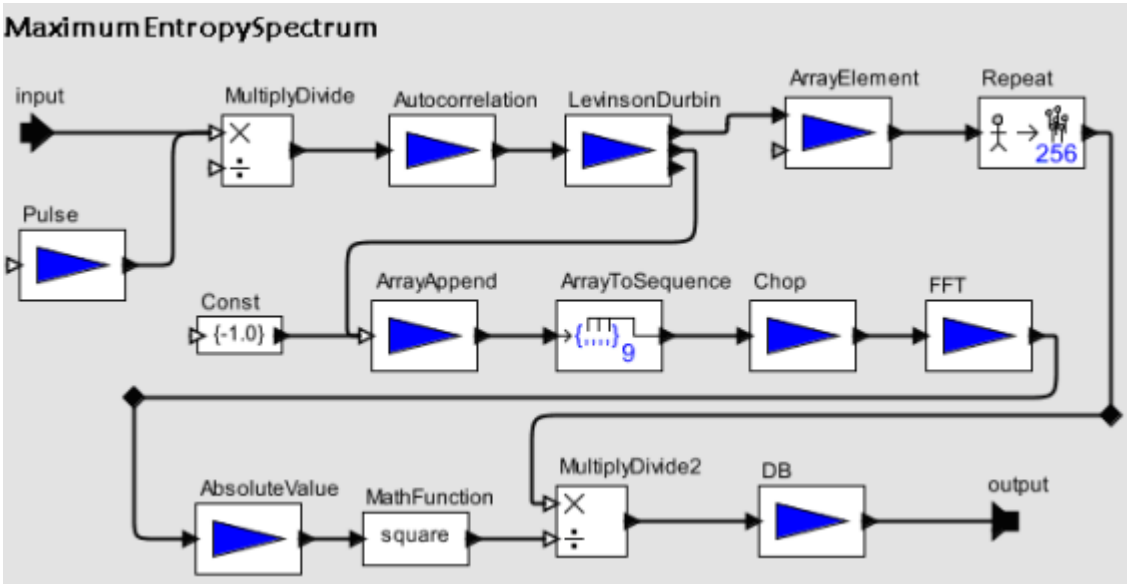



Spectrum
input
Waveform
Multiply
FFT
AbsoluteValue0
Scale
1.0/sq...
DB
output

B.2 Maximum Entropy Spectrum Application from Ptolemy II






MaximumEntropySpectrum
input
MultiplyDivide
Autocorrelation
LevinsonDurbin
ArrayElement
Repeat
256
Pulse
Const
{-1.0}
ArrayAppend
ArrayToSequence
9
Chop
FFT
AbsoluteValue
MathFunction
square
MultiplyDivide2
DB
output

Appendix C: DIF Representations

C.1 Spectrum Application DIF Representation

```
sdf spectrum{
   topology {
         nodes = INPUT, FFT, ABS, SCALE, DB;
         edges = e1(INPUT, FFT), e2(FFT, ABS), e3(ABS, SCALE), e4(SCALE, DB);
   }

   interface {
         outputs = out: DB;
   }

   production {
      e1=1; e2=256; e3=1; e4=1; out = 256;
   }

   consumption {
         e1=256; e2=1; e3=1; e4=1;
   }

   attribute datatype {
      e1 = "double";
      e2 = "Complex";
      e3 = "double";
      e4 = "double";
      out = "double";
   }

   actor INPUT {
         computation = "randomInputGenerator";
         output: OUTPUT = e1;
         executionTime = 660;
   }

   actor FFT {
         computation = "fft";
         input: INPUT = e1;
         order = 8;
         output: OUTPUT = e2;
         executionTime = 6620872;
   }

   actor ABS {
         computation = "absoluteValue";
         input: INPUT = e2;
         output: OUTPUT = e3;
         executionTime = 279;
   }
```

```
    actor SCALE {
            computation = "scale";
            input: INPUT = e3;
            order = 8;
            output: OUTPUT = e4;
            executionTime = 4197;
    }

    actor DB {
            computation = "db";
            input: INPUT = e4;
            min: PARAMETER = -100;
            output: OUTPUT = out;
            executionTime = 118;
    }
}
```

C.2 Maximum Entropy Spectrum DIF Representation

```
sdf MSE {
   topology {
         nodes = INPUT, AUTOCOR, LEVINSON, ARRAYELEMENT, REPEAT,
ARRAYAPPEND, CHOP, FFT, ABS, SQUARE, MULDIV, DB;
         edges = e1(INPUT, AUTOCOR),e2(AUTOCOR, LEVINSON),e3(LEVINSON,
ARRAYELEMENT), e4(LEVINSON, ARRAYAPPEND), e6(ARRAYELEMENT, REPEAT),
e7(REPEAT, MULDIV), e9(ARRAYAPPEND, CHOP), e10(CHOP, FFT), e11(FFT,
ABS), e12(ABS, SQUARE), e13(SQUARE, MULDIV), e14(MULDIV, DB);
   }

   interface {
         outputs = out: DB;
   }

   parameter {
      myConstant : "double" = [-1.0];
   }

   production {
         e1=256; e2=64; e3=33; e4=32; e6=1; e7=256; e9=33; e10=256; e11=256;
e12=256; e13=256; e14=256; out = 256;
   }

   consumption {
         e1=256; e2=64; e3=33; e4=32; e6=1; e7=256; e9=33; e10=256; e11=256;
e12=256; e13=256; e14=256;
   }

   attribute datatype {
         e1 = "double";
         e2 = "double";
         e3 = "double";
         e4 = "double";
         e6 = "double";
         e7 = "double";
         e9 = "double";
         e10 = "double";
         e11 = "Complex";
         e12 = "double";
         e13 = "double";
         e14 = "double";
         out = "double";
   }

   actor INPUT {
         computation = "randomInputGenerator";
         output: OUTPUT = e1;
         executionTime = 660;
   }
```

```
actor AUTOCOR {
        computation = "autocorrelation";
        input: INPUT = e1;
        numberOfInputs = 256;
        numberOfLags = 32;
        biased = 1;
        symmetric = 0;
        output: OUTPUT = e2;
        executionTime = 2535953;
}

actor LEVINSON {
        computation = "LevinsonDurbin";
        input: INPUT = e2;
        autocorrelationValueLength = 64;
        output1: OUTPUT = e3;
        output2: OUTPUT = e4;
        executionTime = 437643;
}

actor ARRAYELEMENT {
        computation = "arrayElement";
        input: INPUT = e3;
        index = 8;
        output: OUTPUT = e6;
        executionTime = 23;
}

actor REPEAT {
        computation = "repeat";
        input: INPUT = e6;
        repetitions = 256;
        output: OUTPUT = e7;
        executionTime = 8475;
}

actor ARRAYAPPEND {
        computation = "arrayAppend";
        constant = myConstant;
        constantLength = 1;
        input: INPUT = e4;
        output: OUTPUT = e9;
        executionTime = 1155;
}

actor CHOP {
        computation = "chop";
        input: INPUT = e9;
        numberToRead = 9;
        numberToWrite = 256;
        offset = 0;
        usePast = 0;
        output: OUTPUT = e10;
        executionTime = 10967;
}
```

```
    actor FFT {
        computation = "fft";
        input: INPUT = e10;
        order = 8;
        output: OUTPUT = e11;
        executionTime = 6620872;
    }

    actor ABS {
        computation = "absoluteValue";
        input: INPUT = e11;
        output: OUTPUT = e12;
        executionTime = 279;
    }

    actor SQUARE {
        computation = "square";
        input: INPUT = e12;
        output: OUTPUT = e13;
        executionTime = 64;
    }

    actor MULDIV {
        computation = "mul_div";
        input1: INPUT = e7;
        input2: INPUT = e13;
        output: OUTPUT = e14;
        executionTime = 141;
    }

    actor DB {
        computation = "db";
        input: INPUT = e14;
        min: PARAMETER = -100;
        output: OUTPUT = out;
        executionTime = 118;
    }
}
```

References and Bibliography

[1] J. R. Agre, L. P. Clare, G. J. Pottie, and N. Romanov, “Development Platform for Self-Organizing Wireless Sensor Networks,” In *Proceedings of SPIE’s 13th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls (AeroSense), Unattended Ground Sensor Technologies and Applications Conference*, April 1999.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, “Wireless sensor networks: a survey,” *Computer Networks: The International Journal of Computer and Telecommunications Networking*, v.38 n.4, p.393-422, March 2002.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(2):151-166, June 1999.

[4] K. J. Binkley and M. Hagiwara, “Particle swarm optimization with area of influence: increasing the effectiveness of the swarm,” In *Proceedings of the Swarm Intelligence Symposium, 2005 (SIS 2005)*, p.p. 45-52. June 2005.

[5] R. C. Eberhart and J. Kennedy, “A New Optimizer Using Particles Swarm Theory,” In *Proceedings of Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 1995.

[6] R. C. Eberhart and Y. Shi, “Particle swarm optimization: developments, applications and resources,” In *Proceedings of the congress on evolutionary computation*, Seoul, Korea, 2001.

[7] R. C. Eberhart and Y. Shi, “Tracking and optimizing dynamic systems with particle swarms,” In *Proceedings of the 2001 Congress on Evolutionary Computation*, CEC2001. IEEE Press (2001) 94 -100.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity — the Ptolemy approach. *Proceedings of the IEEE*, January 2003.

[9] J. Hill and D. Culler, “Mica: a wireless platform for deeply embedded networks,” *IEEE Micro*, 22(6): 12-24, November 2002.

[10] C. Hsu, M. Ko, and S. S. Bhattacharyya. “Software synthesis from the dataflow interchange format,” In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37-49, Dallas, Texas, September 2005.

[11] C. Hsu and S. S. Bhattacharyya. Dataflow interchange format version 0.2. Technical Report UMIACS-TR-2004-66, Institute for Advanced Computer Studies, University of Maryland at College Park, November 2004. Also Computer Science Technical Report CS-TR-4624.

[12] X, Hu and R.C. Eberhart, “Multiobjective optimization using dynamic neighborhood particle swarm optimization,” In *Proceedings of 9th IEEE Cong. Evol. Comput.*, pp. 1677-1681, 2002.

[13] S. Jin, M. Zhou, and A. S. Wu, “Sensor network optimization using a genetic algorithm.” In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, FL, July 2003.

[14] J. Kennedy, “The Particle Swarm: Social Adaptation of Knowledge,” In *Proceedings of IEEE International Conference on Evolutionary Computation*, IEEE Service Center, Piscataway, NJ, 303-308.

[15] J. Kennedy and R. C. Eberhart, “Particle Swarm Optimization”, *IEEE International Conference on Neural Networks*, Perth, Australia, 1995.

[16] M. Kohvakka, M. Hännikäinen, and T. D. Hämäläinen, “Wireless Sensor Prototype Platform.” In *IECON'03: The 29th Annual Conference on the IEEE Industrial Electronics Society*, vol. 2, pp. 1499-1504, November 2003.

[17] LINX Technologies, http://www.linxtechnologies.com, *SCSeries Transceiver, TR-916-SC-P Reference Manual*.

[18] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan, “An Architecture for a Power-Aware Distributed Microsensor Node.” In *IEEE Workshop on Signal Processing Systems*, pp. 581-590, Octobor 2000.

[19] K. E. Parsopoulos and M. N. Vrahatis, “Particle swarm optimization method in multiobjective problems,” In *Proceedings of 2002 ACM Symp. Applied Computing (SAC 2002)*, pp. 603-607, Madrid, Spain, 2002.

[20] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava, “Energy-aware wireless microsensor networks,” *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 40-50, March 2002.

[21] Y. H. Shi and R. C. Eberhart, “A Modified Particle Swarm Optimizer,” In *IEEE International Conference on Evolutionary Computation*, Anchorage, Alaska, May 1998.

[22] Y. Shi and R. C. Eberhart, “Parameter selection in particle swarm optimization,” *Evolutionary Programming VII: Proc. EP 98*. pp. 591-600, Springer-Verlag, New York, 1998.

[23] M. Singh and V. K. Prasanna, “System Level Energy Tradeoffs for Collaborative Computation in Wireless Networks,” *IEEE Workshop on Integrated Management of Power Aware Communications, Computing, and Networking*, May 2002.

[24] S. Sriram, S. S. Bhattacharyya, Embedded Multiprocessors: Scheduling and Synchronization, Marcel Dekker, Inc., 2000.

[25] M. Steinbrecher, S.M. Guru, S. Halgamuge, and R. Kruse, “Optimisation of Energy Usage for Deployment of Sensor Networks.”

[26] Texas Instruments Inc., http://www.ti.com/msp430, *MSP430 Microcontroller, MSP430F1611 Reference Manual*.

[27] J. C. Tillett, R. M. Rao, Sahin, and T. M. Rao, “Particle swarm optimization for the clustering of wireless sensors,” In *Proceedings of SPIE, vol. 5100: Digital Wireless Communications V*, pp. 73-83, 2003.

[28] K. Veeramachaneni and L. A. Osadciw, “Dynamic Sensor Management Using Multi Objective Particle Swarm Optimization,” In *Proceedings of SPIE Defence and Security Symposium*, Orlando, Florida, April 2004.