DYNAMIC SOFTWARE RECONFIGURATION IN SENSOR NETWORKS

By

Sachin V. Kogekar

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December, 2004

Nashville, Tennessee

Approved:

Professor Xenofon Koutsoukos

Professor Aniruddha Gokhale

Dedicated to my parents Radhika and Vijay Kogekar

ACKNOWLEDGMENTS

I would like to acknowledge the partial financial support of Xerox and PARC for this research. I would like to thank my graduate advisor, Dr. Xenofon Koutsoukos, for motivating me and for helping me focus on my research. I appreciate his constant encouragement and guidance, which has made a significant contribution towards the completion of this thesis. I am also very grateful to Dr. Sandeep Neema for his guidance and help in resolving my queries.

I would also like to thank Mr. Brandon Eames and Dr. Aditya Agrawal for always sparing their time to answer my endless queries.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Major advances in wireless communications and electronics technology have resulted in the emergence of low cost, low-power micro-sensors, actuators, embedded processors, and radio devices. Advances in VLSI technology have enabled the integration of all these technologies on a single device with a very small form factor. As a result new sets of computing devices with sensing, actuating, communicating, and processing capability are being developed. Our ever increasing appetite to know more about and be aware of relevant happenings in our surrounding environment is driving the use of these devices in autonomous networks. Such networks are capable of reporting to us after monitoring, sensing and even actuating their environment.

These devices, referred to as wireless sensor nodes, will eventually be deployed in hundreds and thousands in wireless sensor networks. The emergence of such networks has opened up a lot of avenues for research in different fields such as distributed computing, sensor node design, communication technology, operating systems, communication protocols, and software reconfiguration. This thesis addresses the problem of carrying out dynamic software reconfiguration in sensor networks. Applications deployed on sensor networks are composed of many software components executing on individual sensor nodes. The objective is to perform software reconfiguration on individual sensor nodes to alter their functionality without human interference.

1.1 Sensor Networks and their Applications

A wireless ad hoc sensor network consists of a number of nodes spread across a geographical area [1]. The nodes are typically fitted with sensors (for example light, temperature, pressure, and audio sensors) for monitoring their environment and have wireless communication capability. The nodes also have some degree of intelligence for processing the data gathered by their sensors and the capacity to do so. The network is deployed over a geographical area in an ad hoc manner. Some nodes within the network could be placed at known locations, but by and large, the nodes figure out their positions themselves using various localization algorithms [43], [44], [45]. They can also be fitted with actuators that perform some mechanical action. The nodes communicate with each other using wireless technologies like radio. In typical deployments, not all nodes are within wireless range of each other. For the sensor network application to work as a whole, this poses a significant problem. This problem is addressed by the use of various ad hoc routing protocols. These protocols enable two out of range nodes to communicate with each other through some intermediate node.

The basic goals of a wireless ad hoc sensor network can be broadly defined as (1) determining the value of some parameter at a given location, (for example - light, humidity, pressure, temperature) (2) detecting the occurrence of events of interest and estimating parameters of the detected event or events, (3) classifying a detected object, and (4) tracking an object. All these tasks require the proper reporting of the data to the end users. In some cases, there are fairly strict time requirements on this communication.

Major requirements that need to be addressed to facilitate the widespread use of sensors are scalability (sensor networks with over 10,000 to 100,000 nodes are

envisioned), low power consumption, network self-organization, collaborative signal processing, and querying ability.

The sensor networks mentioned can be applied to a lot of problems, in both military and commercial domains. Some of the applications that have been envisioned and in some cases even implemented are:

- **Habitat Monitoring**: Sensor networks can be used in the study of sensitive environmental zones and wildlife habitats. Studies have shown that such habitats react unfavorably to human interference. Sensor nodes connected to various sensors like light, temperature, humidity are spread out in the area of interest, like bird colonies and nests and the data collected is used to study the environment and detect any changes [2], [3], [4].
- **Tracking**: This involves the tracking of a mobile target using a number of sensor nodes [5].
- **Forecasting**: Various sensors can be used to monitor the environment and structures and predict trends related to weather, pollution, floods, earthquakes, bush-fires or structural damages to buildings [6].
- **Education**: Sensor networks can be used to build interaction-based instruction methods to educate children in kindergarten [9].
- **Smart Home/Office**: Sensor nodes equipped with actuators can be used in homes to set the humidity and temperature in the room according to the individual's preferences [7]. Sensor networks are also widely used in security systems in homes, offices and factories.

- **Shooter Localization**: Sensor nodes deployed in an urban environment can be used to calculate the location of an enemy sniper in real time. When a shot is fired, the nodes sense the *shockwave* and *muzzle-blast* generated by the shot. Calculations are performed on the data sent by the individual sensors and the origin of the shot is determined. Counter-Sniper, Battleground Monitoring Systems and Urban-Warfare are archetypal examples for application of such a system. Vanderbilt University has developed a Shooter-Localization system for locating enemy snipers [8].

1.2 Software Reconfiguration

The sensor network applications that are being envisioned will be deployed over large numbers (10,000 to 100,000) of sensor nodes. Often these networks will be deployed in inhospitable and inaccessible terrain. A sensor network composed of many autonomous nodes, exposed to the elements, communicating via unreliable wireless technology is vulnerable to failure. Nodes may fail either from lack of energy or from physical destruction, and new nodes may join the network. The communication between the nodes may be disrupted by noise in parts of the network and environment. A sensor network can be made robust enough to face these challenges if it is able to reconfigure itself. Once the network has been deployed in the field, reconfiguration for the most part applies to software reconfiguration. This involves reconfiguring the software components executing on individual nodes or in parts of the network to alter their behavior in response to the changing environment.

A major challenge to autonomous sensor networks is the limited availability of power. Consider a sensor network composed of strategically placed sensors connected to

cameras to monitor a particular street. Consider an application where a particular person or vehicle is the object of interest. An application that requires all the sensors to be active even when there was no one on the street would waste a lot of power. The application could be designed to activate all sensors only when there was activity in the street. In periods of no activity, only a few of the sensors would be active. Such an application can be implemented if it were possible to reconfigure the components executing on the sensor nodes. Such capability would also be required if some of the sensors were rendered useless due to a mobile obstruction like a big van. The application executing on other sensor nodes would be reconfigured to compensate for the loss of data from the obstructed sensors. Similar examples demonstrating the utility of reconfiguration can be found in almost all applications of sensor networks.

1.3 Software Reconfiguration Problem

Software reconfiguration is one of the major challenges facing sensor networks. It is difficult to estimate all operating conditions for a particular sensor network deployed in any dynamic environment and it is impossible to hardcode every response. This accentuates the need for designing a software reconfiguration architecture for sensor networks that is capable of changing the behavior of the system by switching components executing on individual sensor nodes. The choice of the correct components cannot be enumerated for every operating condition. This choice needs to be resolved during runtime.

The collection of components that can be used in the application forms the design space of the application. Dynamic software reconfiguration involves the resolution of the

component choice during run-time by choosing the components from the design space that satisfy the operating conditions defined in response to stimuli produced by a changing environment to adapt the network's behavior. Dedicated software components called Monitors monitor the behavior of the environment and the sensor network. These are either embedded in the application, or deployed in a stand-alone manner on individual sensor nodes. The process of component choice resolution is dependent upon input from various sensor nodes and is a very computation intensive process that cannot be performed on individual sensor nodes. The implementation of the software reconfiguration architecture requires the implementation of monitoring components that monitor the state of the sensor network which is affected by the changing environment, switching components that reconfigure application components on individual sensor nodes, design space exploration tools that choose the proper components to deploy on particular nodes, and a communication infrastructure that propagates changes to the application configuration to individual sensor nodes.

1.4 Thesis Organization

In this thesis, I present an approach for performing dynamic software reconfiguration on sensor networks. The next chapter presents a summary of related work. Chapter 3 explains the sensor network test-bed. Chapter 4 gives an overview of the software reconfiguration approach. Model Integrated Computing [10] (described in Chapter 5) is used in this approach to capture the sensor network application in explicit models, which can then be manipulated in a user-friendly manner. Chapter 5 describes the modeling paradigm for reconfigurable sensor network applications. Chapter 6

describes Aislemonitor, the sensor network application that performs one dimensional tracking of people walking in an aisle. This application is utilized to demonstrate the proposed software reconfiguration approach. Chapter 7 gives the implementation details of the entire software reconfiguration architecture. Chapter 8 discusses the results of the experiments performed over the sensor network testbed using the Aislemonitor application. Chapter 9 describes the results obtained after evaluating the reconfiguration architecture over a sensor network comprising Berkeley MICA motes [11] simulated in TOSSIM [12]. Chapter 10 contains the conclusion and a discussion of the future work.

CHAPTER II

RELATED WORK

The solution to the problem of dynamic software reconfiguration in sensor networks is closely tied to research in other fields like sensor networks and routing protocols. The first section in this chapter describes wireless sensor networks. The approach proposed in this thesis is based on Model Integrated Computing (MIC). MIC is described in the second section. The third section discusses reconfiguration approaches in various domains. The reconfiguration approach proposed in this thesis utilizes a design space exploration tool called DESERT to resolve the design choices during runtime [36] [40]. This tool is described in the fourth section of this chapter. The proposed software reconfiguration approach is based on the Asynchronous Data Flow model of computation and this is described in the fifth section. The use of the Kalman Filter and tracking mechanisms are discussed in the last section of this chapter.

2.1 Wireless Sensor Networks

Traditionally, sensor networks have been designed with relatively small number of sensors, wired to a central processing unit. However, recent advances in technology and the easy availability of low-power micro-sensors, actuators, embedded processors, and radios are enabling the application of distributed wireless sensing to a wide range of applications [13]. This section explores some of the work gone into the development of wireless sensor networks and their potential applications.

We have been using wired sensor networks for a long time. However, such networks are costly in terms of installation and maintenance. Wired sensor networks are not suitable to monitor large spaces such as factories, eco-systems and battlefields. Such domains are too dynamic for specifically laid out wired networks. Some domains are even dangerous and not suitable for human habitation. Often, the exact location of the phenomenon of interest is unknown. In such environments, the use of distributed wireless sensors with processing capability yields a higher signal to noise ratio (SNR), improves the chances of line of sight, and reduces the effect of environmental obstructions. Such sensors process data in the field and only transmit results of interest to the base station. They do not impose any restrictions on the environment that they monitor and can function without any infrastructure (electric wiring, wired communication channels, power outlets). Conventional methods for monitoring do not scale to modern man-made and natural environments. A different approach is required to monitor such environments.

2.1.1 Application Domains

Wireless sensor networks can be used in a wide variety of domains. Some of the potential areas are physiological monitoring, environmental monitoring, condition based maintenance, smart spaces, military, precision agriculture, transportation, factory instrumentation, and inventory tracking [13].

The use of sensor networks for military operations in urban terrain is described in [14]. An ad hoc sensor network comprising tiny sensor devices called “motes” [11] is deployed in an urban environment. The goal of the sensor network is the surveillance and tracking of friendly, hostile and non-combatant personnel in the urban environment. The

system uses acoustic ranging to determine inter-mote distances and then uses multi-lateration to compute the position of each specific mote. The tracking system starts with the estimation of the position of each mote. Sensors in each mote estimate the distance to a target; all the sensor estimates are then combined using a multi-lateration algorithm to estimate the position of the target. Finally, the sequence of target position estimates is smoothed using a Kalman Filter run on the base station. A Situational Awareness Human Interface system on the base station collects all the target readings from each mote filed into a common situational awareness operating picture that is made available through map-based display systems run on a variety of display systems. This kind of an application will typically involve a large number of sensors.

An application for monitoring the habitat of seabirds on the Great Duck Island in Maine is described in [3]. It is often found that human interference even for the purpose of research and monitoring causes an unfavorable impact on the subjects of interest. Research in Maine suggests that even a 15 minute visit to a cormorant colony can result in up to 20% mortality among eggs and chicks in a given breeding year. Repeated interference can even lead to the abandonment of the habitat and a shift to an unsuitable habitat. Sensor networks offer a viable alternative over traditional invasive methods of monitoring. Sensors can be deployed during sensitive periods such as the onset of the breeding season for studying animals or while plants are dormant or the ground is frozen for carrying out botanical studies. The results of wireless sensor-based monitoring efforts can be compared with previous studies that have traditionally ignored or discounted disturbance effects. Sensor network deployment may even turn out cheaper for conducting long-term studies than traditional personnel-rich methods. In the traditional

approach a substantial proportion of logistics and infrastructure are devoted to the maintenance of field studies, often at some discomfort and occasionally at some real risk. A “deploy ’em and leave ’em” strategy of wireless sensor usage would limit logistical needs to initial placement and occasional servicing. The use of wireless sensors could also greatly increase access to a wider array of study sites, often limited by concerns about frequent access and habitability.

The sensor network applications described in the previous paragraphs can be made more efficient and tolerant to failure through the use of reconfigurable software components. My work focuses on building a software reconfiguration infrastructure that utilizes the modular nature of the applications to enhance their ability to adapt to changing operating conditions.

2.1.2 Sensor Node Technology

One of the most widely used platforms for researching wireless sensor networks with limited resources is the Berkeley MICA mote [11] (Figure 1), designed at the University of California, Berkeley. The MICA mote has a 4MHz microcontroller, 4KB of RAM, 128KB of flash memory and a 916 MHz wireless radio transceiver with a transfer rate of 19.2 Kbps and range of 200 feet. It has a very small form factor (58mm X 32mm X 7mm) and is powered by two AA batteries. Daughtercards with various sensors and actuators are available, including photo, temperature, humidity, infrared and barometric pressure sensors, accelerometers, magnetometers, microphones, and sounders.

The Berkeley MICA motes run the TinyOS operating system [11], an open source, event driven and modular OS designed to be used with networked sensors. A

TinyOS application is a statically compiled graph of components. Components have memory frames to store their state, and communicate with each other through used and provided interfaces that contain logically related commands and events [15]. Components can post tasks to process longer running computations, which are executed in order by the scheduler. TinyOS comes with a library of OS components that handle task scheduling, radio communication, clocks and timers, ADC, I/O and EEPROM abstractions, and power management. Application developers can select a subset of these modules, extend or override them if necessary, and statically compile them into the final executable. A typical MICA system consists of tens to hundreds of motes forming an ad hoc multi-hop network and a base station that is typically a PC class computer.

Figure 1 - The MICA2 Mote

Stargate [41] is a powerful single board sensor with communications and sensor signal processing capabilities. The Stargate uses Intel's® latest generation 400MHz X-Scale® processor (PXA255). In addition to traditional single board computer

applications, the Stargate directly supports applications designed around Intel's Open-Source Robotics initiative as well as TinyOS-based wireless sensor networks.

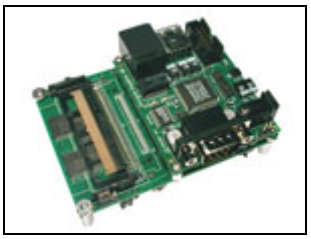

Figure 2 – Stargate [41]

Another powerful, yet compact sensor node is the OpenBrick [32] device. It is small (220 x 165 x 42 mm), light (about 1200 g), and includes three (3) RJ45 LAN connectors. It has a fan-less 533 Mhz x86 compatible VIA C3 processor and 256 MB SDRAM. Software can be installed on a Compact Flash or on a Hard Disk. It also has USB 2.0 connectivity. The device supports the Linux operating system. OpenBrick E is part of the sensor network testbed described in Chapter 3.

The sensor nodes do not have enough resources to evaluate the QoS parameters, search for the next configuration and compute the necessary reconfiguration steps. They can, however, communicate the measured parameters to the base-station where the computationally intensive reconfiguration decisions are made and the necessary elementary reconfiguration commands are sent back to the sensor nodes that execute them [16].

2.1.3 Routing Protocols

Wireless sensor networks can be broadly classified into two varieties based on their wireless configuration. Wireless networks with fixed and wired gateways are called Infrastructured networks. The bridges for these networks are known as *base stations*. A node within these networks connects to, and communicates with, the nearest base station that is within its communication radius. If a node travels out of range of one base station and into the range of another, a “handoff” occurs from the old base station to the new, and the node is able to continue communication seamlessly throughout the network. The second type of mobile wireless network is the infrastructureless mobile network, commonly known as an *ad hoc network*. Infrastructureless networks have no fixed routers; all nodes are capable of movement and can be connected dynamically in an arbitrary manner. Nodes of these networks function as routers which discover and maintain routes to other nodes in the network.

In order to facilitate communication within ad hoc networks, a routing protocol is used to discover routes between nodes. The primary goal of such an ad hoc network routing protocol is correct and efficient route establishment between a pair of nodes so that messages may be delivered in a timely manner. Route construction needs to be done with a minimum of overhead and bandwidth consumption. Different existing ad hoc routing protocols are discussed and compared in [17]. Existing routing protocols for ad hoc networks can be categorized as (1) Table-driven and (2) Source-initiated (demand-driven) as shown in Figure 3 [17].

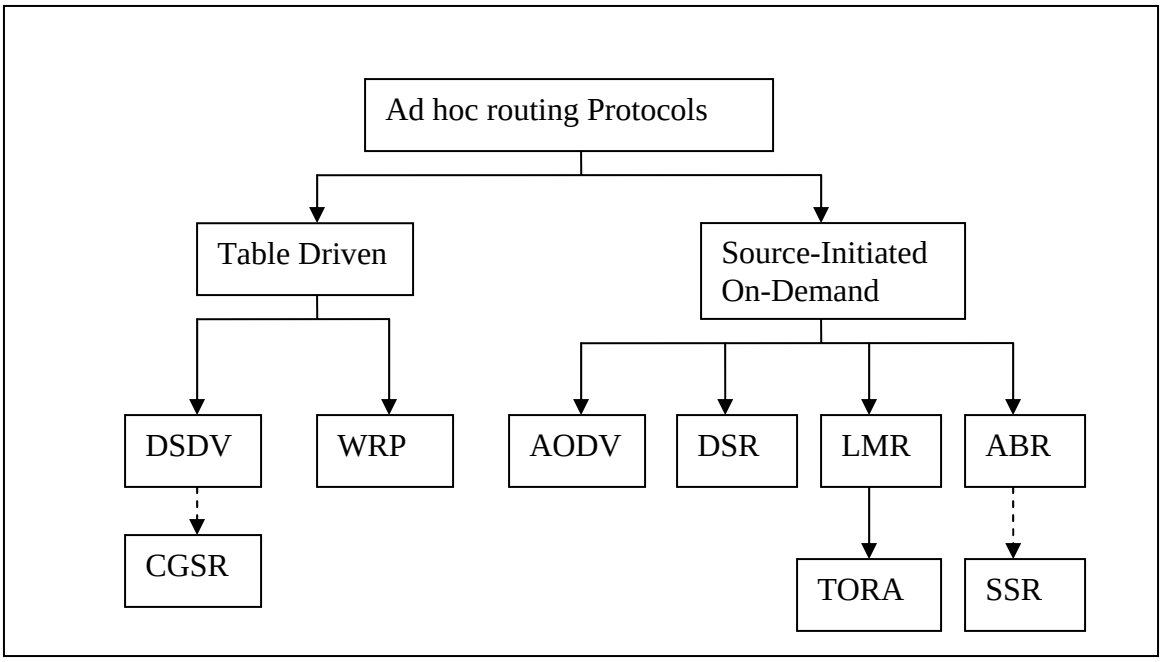

Figure 3 - Categorization of Ad hoc Networks [17]

National Institute of Standards and Technology's (NIST) implementation of the Ad hoc On-Demand Distance Vector (AODV) [18] routing protocol is deployed on our sensor network testbed. When a source node desires to send a message to some destination node and does not already have a valid route to that destination, it initiates a *path discovery* process to locate the other node. It broadcasts a route request (RREQ) packet to its neighbors, which then forward the request to their neighbors, and so on, until either the destination or an intermediate node with a "fresh enough" route to the destination is located. AODV utilizes destination sequence numbers to ensure all routes are loop-free and contain the most recent route information. The source node includes in the RREQ the most recent sequence number it has for the destination. Intermediate nodes can reply to the RREQ only if they have a route to the destination whose corresponding destination sequence number is greater than or equal to that contained in the RREQ.

Once the RREQ reaches the destination or an intermediate node with a fresh enough route, the destination/intermediate node responds by unicasting a route reply (RREP) packet back to the neighbor from which it first received the RREQ. As the RREP is routed back along the reverse path, nodes along this path set up forward route entries in their route tables, which point to the node from which the RREP came. These forward route entries indicate the active forward route. Associated with each route entry is a route timer, which will cause the deletion of the entry if it is not used within the specified lifetime. Because the RREP is forwarded along the path established by the RREQ, AODV only supports the use of symmetric links.

If a source node moves, it is able to reinitiate the route discovery protocol to find a new route to the destination. If a node along the route moves, its upstream neighbor notices the move and propagates a *link failure notification* message (an RREP with infinite metric) to each of its active upstream neighbors to inform them of the erasure of that part of the route [18]. These nodes in turn propagate the *link failure notification* to their upstream neighbors, and so on until the source node is reached. The source node may then choose to reinitiate route discovery for that destination if a route is still desired. An additional aspect of the protocol is the use of *hello* messages, periodic local broadcasts by a node to inform each mobile node of other nodes in its neighborhood. Hello messages can be used to maintain the local connectivity of a node. However, the use of hello messages is not required. Nodes may listen for retransmission of data packets to ensure that the next hop is still within reach.

The use of a routing protocol facilitates communication between sensors that are not within wireless range of each other. The routing protocol is an essential part of the

sensor network and enables the network to function as a whole. The AODV routing protocol has been deployed on the experimental testbed to emulate real world sensor networks. The OpenBrick devices communicate with each other using the AODV protocol.

2.2 Model Integrated Computing

Model-Integrated Computing (MIC) [10] employs domain-specific models to represent the software, its environment, and their relationship. Using Model-Integrated Program Synthesis (MIPS) [46], these models are then used to automatically synthesize the embedded applications and to generate inputs to commercial off the shelf (COTS) analysis tools [19]. This approach speeds up the design cycle, facilitates the evolution of the application and helps system maintenance, dramatically reducing costs during the lifecycle of the system. We use MIC to model reconfigurable component based sensor network applications.

The Multigraph Architecture (MGA) is a toolkit for creating domain-specific MIPS environments. The MGA is illustrated in Figure 4. A metaprogramming interface is used to specify the modeling paradigm of the application domain. The modeling paradigm is the modeling language of the domain specifying the objects and their relationships. In addition to syntactic rules, semantic information can also be described as a set of constraints. The Unified Modeling Language (UML) and the Object Constraint Language (OCL), respectively, are used for these purposes in the MGA. These specifications, called metamodels, are used to automatically generate the MIPS environment for the domain. An interesting aspect of this approach is that a MIPS

environment itself is used to build the metamodels [20]. In this approach, the MGA is used to design the metamodel for the reconfigurable applications modeling language SNRAMoLa (described in Chapter 5).

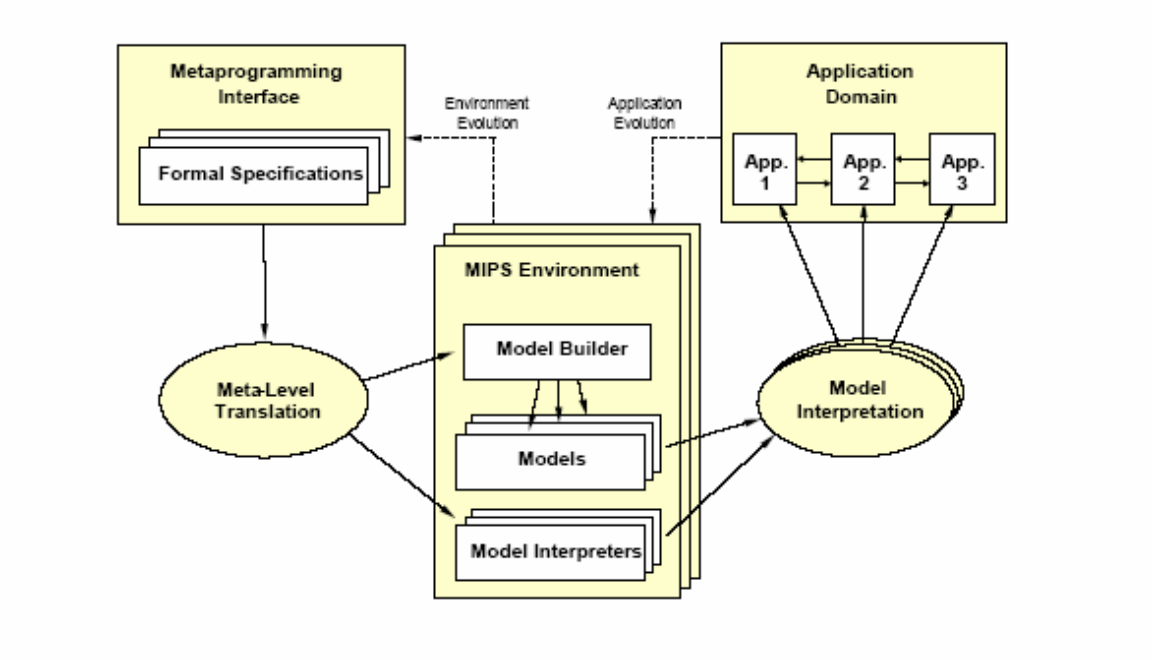

Figure 4 - The Multigraph Architecture

The generated domain-specific MIPS environment is used to build domain models that are stored in a model database. These models are used to automatically generate the applications or to synthesize input to different COTS analysis tools. This translation process is called model interpretation.

In the proposed approach, the sensor network application is modeled in terms of its components and their interactions (explained in Chapter 5). Valid configurations, which include only those components that actually execute in the application, are then

generated. The models are used to express a design choice and produce a valid configuration every time the software reconfiguration process is triggered during runtime.

2.3 Software Reconfiguration

This thesis proposes an architecture to carry out dynamic software reconfiguration in wireless sensor networks. This section explores the work done on this problem in other and related domains.

Software architectural models can be designed to monitor and guide dynamic changes to an application. The work in [21] presents a generic architecture for adaptation in pervasive networks using a client/server networking example. The centerpiece of this approach is the use of stylized architectural models. The architectural model of a pervasive network is represented as a graph of interacting components with nodes in the graph termed as components and arcs termed as connectors, which represent interaction between components. These software architectural models, originally created to support design-time development tools, can be augmented with adaptation operations and repair strategies that apply these operations to adapt the architecture during runtime. The approach in [21] suggests a method to monitor the system using probes and gauges.

An architectural model can be built to represent component-based software systems on graphs. Adaptation and repair of such systems poses a major challenge. Low computational resources on sensor nodes do not allow extensive repair strategies being embedded into application components. An approach for representing these strategies is required.

The work in [22] presents a lightweight infrastructure for managing dynamic reconfiguration called Lira that applies and extends the concepts of network management to component-based, distributed software systems. Two extreme approaches to carry out reconfiguration can be identified as internal and external reconfiguration. Internal reconfiguration relies on the programmer to build reconfiguration facilities directly into components while external reconfiguration relies on some external entity to determine the reconfiguration of the component. Lira is designed to perform both component-level reconfigurations and scalable application-level reconfigurations through the use of agents associated with individual components and a hierarchy of managers. Agents are specifically programmed for individual components to respond to reconfiguration requests appropriate for that component. A Management Information Base (MIB) associated with each component captures its state variables. The agent then uses the MIB to trigger reconfiguration. Managers embody the logic for monitoring the state of one or more components, and for determining when and how to execute reconfiguration activities. A protocol based on Simple Network Management Protocol (SNMP) is used for communication among managers and agents.

In order to implement a reconfiguration approach, the structure and software design of components needs to be developed. Once the structure of components is standardized, various reconfiguration approaches can be tried on the components. Application and reconfiguration components executing on resource limited sensor nodes cannot contain too many complex computations. This thesis uses the term components in relation to software processes, which are relegated to perform atomic tasks that really do not require any internal reconfiguration. However, the application as a whole is

composed of many components and reconfiguration means the execution of selected components. The thesis proposes a reconfiguration agent, much like the Manager, which executes on each individual sensor device. This agent performs the reconfiguration activity on individual nodes by switching components according to the reconfiguration script that it receives from the base station. The 'real' reconfiguration, the selection of appropriate components for execution, in response to a change in the environment, occurs at a central base station. This is necessary in distributed sensor networks because individual sensors typically do not have the capability to perform intense computations, which are needed for design space exploration.

An approach to carry out reconfiguration for fault tolerance by applying graph grammars is suggested in [23]. The work suggests the use of Reconfiguration Graph Grammars (RGG) to target the problem of carrying out dynamic reconfiguration in processor arrays. The nodes of a graph are associated with individual processors of the processor array and the edges are associated with those inter-processor connection lines that are active. Production rules defined in the RGG carry out the reconfiguration in the processor arrays. Consider an example of an array of processors connected to each other. These can be represented in the form of a graph. Each row in the array contains some additional (spare) processors that are not used. When one of the used processor fails, the productions that define the graph grammar for the reconfiguration algorithm dynamically reconfigure the arrays such that one of the spare processors is used in place of the faulty one and the incoming edges (active communication lines) are transferred to the spare processor along with the outgoing edges in the graph. RGG-based reconfiguration can be implemented by using a program in the memory of each processor. A neighboring

processor can carry out the detection of a failed processor. It can even be combined with reconfiguration initiation by the neighboring processor.

The DJINN multimedia-programming framework is designed to support the construction and dynamic reconfiguration of distributed multimedia applications [24]. The main requirements addressed by DJINN are to provide QoS and integrity guarantees for complex multimedia applications, both in their steady state and during reconfigurations. DJINN includes (1) Programming support for distributed multimedia applications, (2) Dynamic reconfiguration, and (3) Support for QoS negotiation, admission control and the specification of integrity constraints. DJINN applications are made up of autonomous 'active' components that produce, consume and transform multimedia data streams and are distributed with the multimedia hardware and 'model' components arranged in a tree-structured hierarchy where the leaves of the tree are atomic model components each corresponding to a single active component. Application programmers do all the development in the model layer. Atomic model components model the QoS characteristics of their underlying active components as sets of linear quadratic relations between attributes. Application integrity is modeled by sets of predicates attached to Model components. The predicates are evaluated in leaf to root order during integrity tests and all must be true for the application's configuration to be considered valid.

A reconfiguration manager is responsible for controlling and validating changes to the application model. Application configuration – and reconfiguration –is expressed in terms of paths: model layer end-to-end management constructs describing the media data flow between a pair of endpoints chosen by the application. A path encapsulates an

arbitrary sequence of ports and intervening components that carry its data. A reconfiguration moves the application from one consistent state to another in an atomic manner. Before new active components are created and started, the model must pass the integrity tests and an admissions test. Each admission test utilizes the application's OoS model, and is performed in three stages: (1) to gather application imposed constraints, (2) to determine constraints on resources, and (3) to generate a solution using a cost-benefit analysis. The approach for solving constraint relations is borrowed from operations research used in optimization problems. These techniques utilize a benefit function to find optimum values for a set of variables given a set of constraints.

In my approach, sensor network applications are modeled as graphs where the nodes represent the components and the arcs represent their interaction. Constraints associated with the application are modeled using Object Constraint Language (OCL). These constraints are then evaluated by using a design space exploration tool, which utilizes Object Binary Decision Diagrams (OBDD) [39].

The use of reconfiguration to implement Automatic Target Recognition (ATR) applications onto field programmable gate arrays (FPGAs) is described in [25]. Bit-level operations that comprise much of the ATR computational burden map extremely efficiently into FPGAs because the specificity of ATR target templates can be leveraged via fast reconfiguration. ATR involves correlation of chips from the observed image with the templates already stored in memory which often causes bottlenecks due to the excess processing loand. FPGAs offer an attractive solution to the correlation problem. The operations being performed occur directly at the bit level and are dominated by shifts and add, making them easy to map into the hardware provided by the FPGA in the form of

adder trees. The templates are hard-wired into the FPGA (in the form of adder trees) while image pixels are clocked past it. Combining multiple templates on a single FPGA can increase efficiency. To minimize the number of FPGA reconfigurations necessary to correlate a given target image against the entire set of templates, it is necessary to maximize the number of templates placed in every configuration of the FPGA. This can be achieved by partitioning the set of templates into groups that can share computations (adder trees) so that fewer resources are used per template. The reconfiguration is thus the problem of choosing the right set of templates to group together on an FPGA. This can be achieved by ensuring that any template added to an existing group is approximately the same size as templates in the group. One of the heuristics used in deciding whether or not to include a template into a newly formed partition is to determine the number of new terms that adding the template would create in the partition's adder tree.

An efficient optimal algorithm for minimizing runtime reconfiguration delay is presented in [26]. A major drawback of using runtime reconfiguration is the significant delay of reprogramming the hardware. In many applications, only a small portion of the design changes at a time and there is no need to reconfigure the entire hardware for instantiating a new design. Partial reconfiguration allows the users to change only part of the design that needs to be updated and hence decreases the reconfiguration time. A provably optimal algorithm to minimize the total delay incurred by partial reconfiguration is presented in [26]. The algorithm outputs an execution order of the operations on hardware resources such that the total runtime reconfiguration is minimized. This thesis does not look into optimizing the reconfiguration delay. However,

the proposed approach does not affect the working of other components in the application and only affects the components that are being acted upon by the reconfiguration agent.

A critical issue for complex component-based systems design is the modeling and analysis of architecture, especially in systems whose architecture changes dynamically (during run time). This is because dynamic changes to architectural structure may interact in subtle ways with on-going computations of the system. It is valuable to provide a modeling approach that accounts for the interactions between architectural reconfiguration and non-reconfiguration systems functionality, while maintaining a separation of concerns between these two aspects of the system [27]. The modeling tool, Wright [27], addresses the problem of capturing dynamic architectures. The key is to use a uniform notation and semantic base for both reconfiguration and steady-state behavior, while at the same time providing syntactic separation between the two. This permits the viewing of the architecture in terms of a set of possible architectural snapshots, each with its own steady-state behavior. Transitions between these snapshots are accounted for by special reconfiguration-triggering events. The proposed reconfiguration approach also expresses reconfigurable and non-reconfigurable components on the same modeling canvas. However, care is taken to separate the modeling of the two. This kind of modeling enables the visualization of varied valid configurations of the applications under changing conditions.

2.4 Design Space Exploration Tool (DESERT)

As indicated in the previous sections, software reconfiguration is a computation intensive process that cannot be typically performed on resource limited sensor nodes. The proposed approach includes a design space exploration tool called DESERT in the reconfiguration architecture. Design space exploration in DESERT [36] entails pruning the design-space with the applied constraints. An intuitive user interface lets the user perform the exploration interactively. The end result of the exploration is a pruned design space that contains only a few design configurations that are valid with respect to the applied constraints.

2.4.1 Design Space Exploration

The design space exploration tool, DESERT, has been developed using MIC. Figure 5 shows the meta-model of the DESERT input modeling language. The core concepts in the modeling language are *Space*-s, *Element*-s, *Property*-s, and *Constraint*-s. An Element represents a hierarchically composed item in the space to be explored. A value of true for Decomposition attribute implies inclusive AND-decomposition, which means that all the children of the Element are included in all configurations. A value of false, on the other hand, implies OR-decomposition, which means that the Element is composed of exactly one of its children in any configuration. The children of an OR-decomposed element represent alternatives, i.e. a choice has to be made among the alternatives in the design space exploration based on constraints. An element with no children represents a leaf in the hierarchy, regardless of the value of its Decomposition attribute. A Space is simply a composition of Elements, and is equivalent to an Element

with a Decomposition value of true. Several Spaces may be composed together to define the aggregate design space for the system.

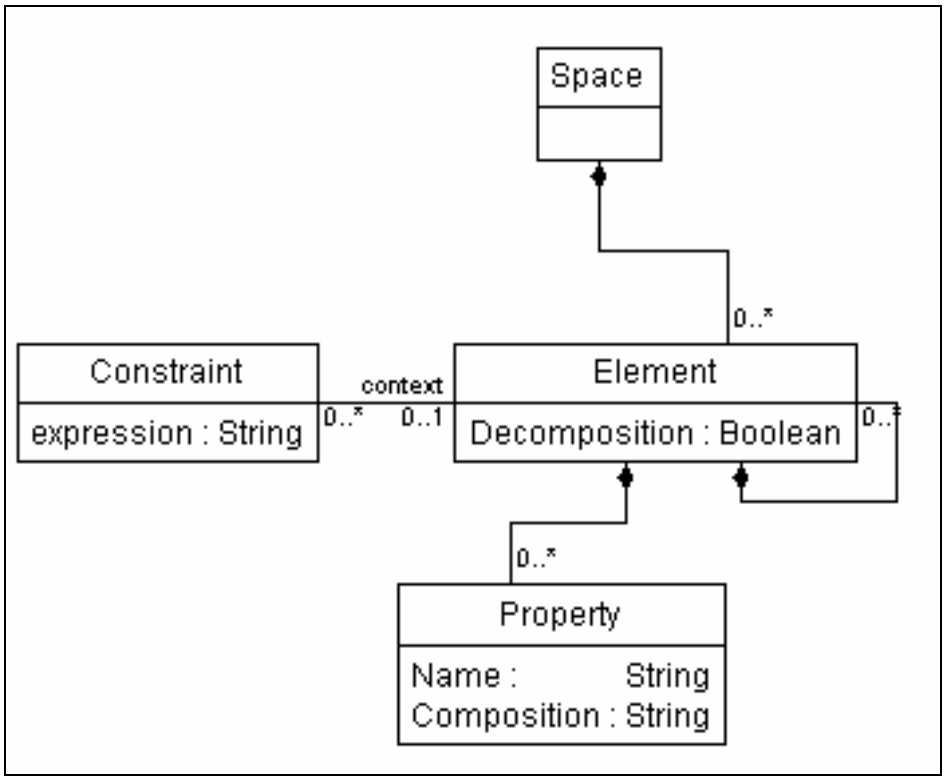

Figure 5 - DESERT Meta-Model

An Element can contain zero or more Properties. The general notion of Property is a characterization of an element; however, the specification and its semantic interpretation may differ based on the decomposition of the element and its placement in the hierarchy. For leaf elements, property values are specified as an input to DESERT, whereas for non-leaf elements, DESERT computes property values, while evaluating constraints, based on the decomposition of the non-leaf element, as well as the Composition policy of the property. Multiple values may be provided for a Property of a leaf element, representing another dimension of choice with a kind of parameterization.

For OR-decomposed elements, the composite property is an exclusive-OR of any one of the child elements, while for AND-decomposed elements the property of the element is a composition per the Composition policy. DESERT implements a number of Composition policies, such as additive, multiplicative, maximum (or minimum). Custom Composition policies are also supported; however, the user is required to provide the composition algorithm. DESERT has well-defined interfaces for implementing custom composition policies.

Constraints are the specification that the design space exploration evaluates over the provided design space, and produces a "pruned" space that contains only such designs that satisfy the constraint. To remain consistent with the selected meta-modeling language (UML class diagrams and OCL), a variant of OCL is used for constraint specification.

This mechanism of structuring design spaces can be summarized as hierarchically layered parameterized alternatives. The following example demonstrates its scalability in representing large design spaces: With a alternative implementations per OR-decomposed element, and n OR-decomposed elements on each level of an m-level deep refinement hierarchy, this representation can define: a^{k_m} design configurations, where $k_m = (k_{m-1}+1)\times n$, and $k_1 = n$, using just $(a \times n)^m$ leaf elements. As an example, with $n = 4$, $a = 3$, and $m = 3$, a total of 1728 leaf elements can represent 3^{84} design configurations in the space!

Formally, a design space is a set and its formulation is demonstrated in the following expressions. A configuration is a particular selection of choices in the space. Let $Configs(d)$ be the set of all configurations that include an element d, and $\chi(d)$ be the

set of children of d. Also let D_j be the set of values of property j, and let $P(l)$ be the set of properties in a leaf element l. Then, the set of possible instantiations $PS(l)$ of the leaf element l can be defined as:

$$PS(l) = \prod_{j}^{P(l)} D_j \qquad \textbf{(1)}$$

The set of configurations can be constructed recursively, depending on element decomposition, as follows:

$$Configs(d) = \begin{cases} PS(d) & \text{LEAF} \\ \prod_{x \in \chi(d)} Configs(x) & \text{AND} \\ \bigcup_{x \in \chi(d)} Configs(x) & \text{OR} \end{cases} \qquad \textbf{(2)}$$

Let, $\Re_k$ be the root element of the k-th space, then $Configs(\Re_k)$ is the set of all configurations in the k-th space. The aggregate design space can now be defined as:

$$DS = \prod_{k} Configs(\Re_k) \qquad \textbf{(3)}$$

2.4.2 Design Space Encoding and Pruning

Manipulation of design-spaces can be reduced to set operations: calculating product spaces (composition of design spaces) and finding subspaces that satisfy various (structural) constraints. Since the size of design-spaces is frequently huge, execution of these set manipulation operations by enumerating all elements is hopeless. Therefore the manipulation operations are performed *symbolically*. Two problems had to be solved: 1) symbolic representation of design-spaces, and 2) symbolic representation of constraints.

If the parameters of model objects are restricted to finite domains, the design space will be also finite. By introducing a binary encoding of the elements in a finite set, all operations involving the set and its subsets can be represented as Boolean functions

[39]. These can then be symbolically manipulated with Ordered Binary Decision Diagrams (OBDD-s) [39], a powerful tool for representing, and performing operations involving Boolean functions. The choice of encoding scheme has a strong impact on the scalability of the symbolic manipulation algorithms, as it determines the number of binary variables required for representing the sets. In addition to encoding the structure of the design-space, the encoding scheme has also to encode the parameters of the parameterized model components. Subsequent to encoding, and deciding the variable ordering, the symbolic Boolean representation is mapped to an OBDD representation in a straightforward manner. The details of this encoding scheme have been described in [40].

There are two basic categories of structural constraints that DESERT can compute efficiently.

Compatibility and Inter-space constraints – These constraints specify relations among subspaces in the overall design space, expressing semantic compatibility between different elements. Symbolically, the constraints can be represented as a Boolean expression over the Boolean representation of the elements of the design-space.

Property constraints – Property constraints specify bounds on the composite properties of elements in the composed system. The important challenge for the property constraints are that they are derived from structural characteristics of designs. As we mentioned earlier different properties compose differently, e.g. cost can be composed additively, latency can be composed as additively for pipelined components, and as maximum for parallel components, etc. DESERT provides some built-in composition functions (addition, maximum, minimum, etc.), and has a well-defined interface for creating custom composition functions

In addition to these basic categories of constraints, complex constraints may be expressed by combining one or more of these constraints with first order logic connectives. The symbolic representation of the complex constraints can be accomplished simply by composing the symbolic representation of the basic constraints.

OBDD based representations scale well for representing the structure of the design space (nested AND/OR expressions). The critical challenge in scalability occurs during the design-space pruning step. Automatic application of complex constraints to large spaces may result in explosion of the OBDD-s therefore DESERT has an interactive user interface to influence this process. Users can control the importance of constraints and select the sequence order of their application [16].

The primary advantage of the symbolic design space pruning approach is that it is exhaustive: the pruned space includes all of the designs, which meet the applied design constraints. A significantly simpler, but still useful alternative approach to design space pruning could be to find a single design configuration (not all), which satisfies the selected design constraints. The controller at the base station selects the first amongst all the valid configurations generated by DESERT.

2.5 Model of Computation

In order to model component based sensor network applications, we need to base our applications on a formal model of computation. A model of computation can be defined as a formal, abstract definition of a computer. Using a model one can more easily analyze the intrinsic execution time or memory space of an algorithm while ignoring many implementation issues [48]. Sensor network applications are typically composed of a number of components or processes that exchange data. We use the Asynchronous Data Flow (ASDF) process network [42] as a base to model our applications.

Under the data flow paradigm, applications are described as directed graphs where the nodes represent computations (or functions) and the arcs represent data paths. The data flow principle is that any node can *fire* (perform its computation) whenever input data is available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, hence the concurrency. Because the program execution is controlled by the availability of data, data flow programs are said to be data driven. The only influence one node has on another is the data passing through the arcs.

In most graphical programming languages, the nodes of the graph can be viewed as processes that run concurrently and exchange data over the arcs of the graph. Dataflow process networks are shown to be a special case of the Kahn process networks in [42]. In a process network, concurrent processes communicate with each other through one-way first-in-first-out (FIFO) channels with unbounded capacity. Each data token is written to the channel exactly once and read from the channel exactly once. Writes to the channel are non-blocking (they always succeed) while reads are blocking. A process that attempts

to read from a channel stalls if the channel is empty. This model of computation does not actually require either multitasking or parallelism, but it is capable of exploiting both. It also does not require infinite queues and can be implemented to use memory much more efficiently. Unlike the Synchronous Data Flow (SDF) [28] model of computation which requires static scheduling, ASDF process networks do not specify any timing information or the rates at which data is generated and nodes fire and can be scheduled dynamically.

This approach proposed in this thesis uses the ASDF model of computation to model and implement sensor network applications. The sensor network applications are designed as a group of independent processes communicating with each other using inter-process communication directives and constructs. This kind of software architecture enables the dynamic switching of components executing in the application. The sensor network application is mapped onto an ASDF graph such that each node in the graph represents a process and the arcs connecting two nodes represent the inter-process communication. Moreover, the processes in the application are data driven and fire only when they receive data from another process. The processes that are not connected to any other process may fire at any time as in the ASDF model.

2.6 Motion Detection/Tracking/Estimation

I implemented a sensor network application called Aislemonitor for single dimensional tracking of people walking in an aisle (described in Chapter 6). This application was deployed on our sensor network testbed (described in Chapter 3) and used to demonstrate our software reconfiguration approach. This section explores the work gone into sensing and tracking objects of interest using sensor networks.

A methodology for planning and controlling the sensing, processing and communication actions needed to accomplish a certain mission using sensor networks while respecting the system resource constraints like power consumption, communication range, bandwidth and susceptibility to noise limitations due to the wireless technologies such as radio links commonly used in sensor devices is described in [29]. Emphasis is laid on techniques to address real world high-level queries such as “who is the leader of the people walking in the building” or “is friendly vehicle *a* surrounded by enemy vehicles”. A mathematical framework on how high-level queries can be transformed into low level sensing, computation and communication operations designed to produce the desired answers, while minimizing the power and other resources expended in satisfying the queries is proposed. Several of such queries refer to global, aggregate and relational aspects of the environment. Though most sensing acquires data in the continuous domain, the information most useful to the system’s clients is often of an aggregated or discreet nature. By its nature sensor network is a hierarchical hybrid system where sampled continuous signals transition to discreet symbolic information as we go up the task hierarchy. The pushing of the interface between continuous and discrete to a very low level in the system architecture is possible and can yield significant benefits in economy

and speed. Emphasis is put on tracking spatial or temporal relations between objects and local or global attributes of the environment than the detailed estimation of positions and poses of individual objects. By focusing on relations and the logical structure of the evidence with respect to the task at hand, it will be possible to allocate the sensor and computation resources where they are most needed.

The tracking algorithm deployed on the sensor network testbed has some constraints; it allows the tracking of only one person per sensor node at any given time. Tracking of multiple people is a more complex problem. The ideas presented in [29] can be used to estimate the number of people walking together in the range of an individual sensor at the same time. A Kalman Filter is used to smooth out the positions of the center of mass of the person walking in the aisle detected by the application.

The Kalman filter [30] [31] provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown [30]. In this approach, The Kalman Filter is used to estimate the position of a person in an aisle. When a person is walking in the aisle, his/her observed position is fed into the Kalman Filter. The filter then computes the person's position and speed.

A Kalman filter is simply an optimal recursive data processing algorithm. One aspect of this optimality is that it incorporates all information that can be provided to it. It processes all available measurements, regardless of their precision, to estimate the current value of the variables of interest, with use of (1) knowledge of the system and measurement device dynamics, (2) the statistical description of the system noises,

measurement errors and uncertainty in the dynamics models, and (3) any available information about the initial conditions of the variables of interest. For example, to determine the velocity of an aircraft, one could use a Doppler radar or the velocity indications of an inertial navigation system, or the pilot and static pressure and relative wind information in the air data system. Rather than ignore any of these outputs, a Kalman filter could be built to combine all this data and knowledge of the various systems' dynamics to generate an overall best estimate velocity.

The word recursive means that, unlike certain data processing concepts, the Kalman filter does not require all the previous data to be kept in the storage and reprocessed every time a new measurement is taken. This is of vital importance to the practicality of filter implementation.

The filter is actually a data processing algorithm. Despite the typical notation of a filter as a black box containing electrical networks, the fact is that in most practical applications, the filter is just a computer program in a central processor. As such, it inherently incorporates discrete time measurement samples rather than continuous time inputs. Figure 6 depicts a typical situation in which a Kalman filter could be used advantageously. A system of some sort is driven by some known controls, and measuring devices provide the value of certain pertinent quantities. Knowledge of these system inputs and outputs is all that is explicitly available from the physical system for the estimation process.

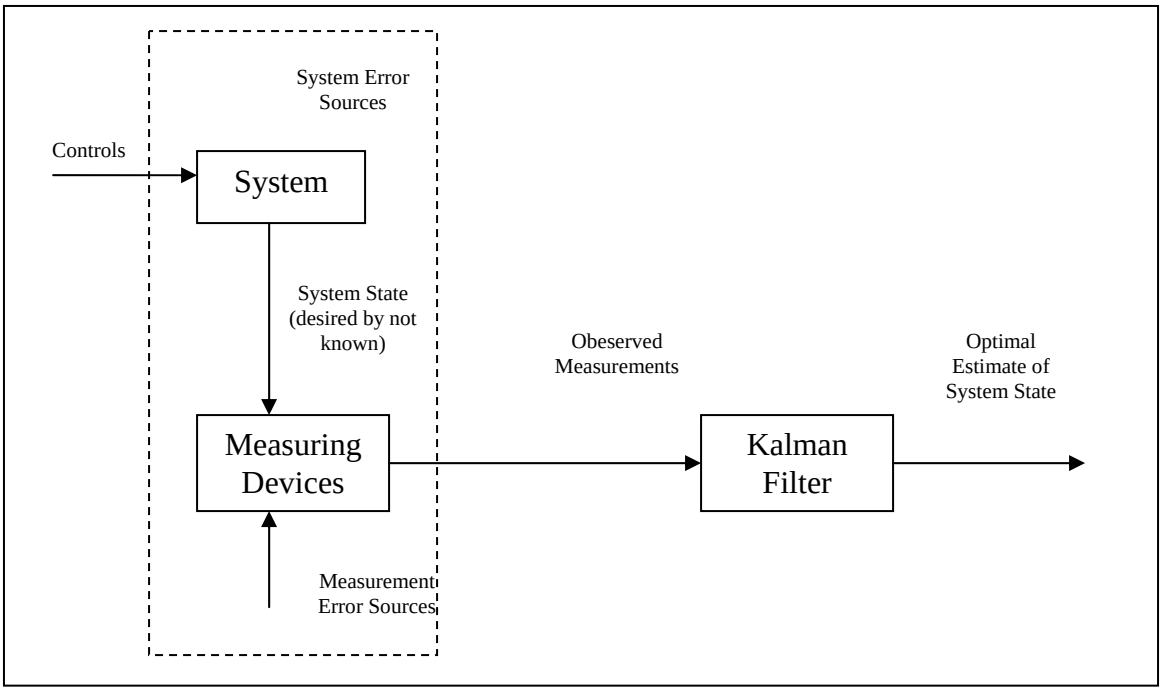

Figure 6 - Typical Kalman Filter Application

Often the variables of interest, some finite number of quantities to describe the "state" of the system, cannot be measured directly, and some means of inferring these values from the available data must be generated. For instance, an air data system directly provides static and pitot pressures, from which velocity must be inferred. This inference is complicated by the facts that the system is typically driven by inputs other than our known controls and that the relationships among the various "state" variables and measured outputs are known only with some degree of uncertainty. Furthermore, any measurement will be corrupted to some degree by noise, biases, and device inaccuracies, and so a means of extracting valuable information from a noisy signal must be provided as well. There may also be a number of different measuring devices, each with its own particular dynamics and error characteristics that provide some information about a particular variable, and it would be desirable to combine their outputs in a systematic and

optimal manner. A Kalman filter combines all the available measurement data, plus prior knowledge about the system and measuring devices, to produce an estimate of the desired variables in such a manner that the error is minimized statistically.

The filter performs conditional probability density propagation for problems in which the system can be described through a *linear* model and in which the system and measurement noises are *white* and *Gaussian.* Under these conditions, the mean, mode, median and virtually any reasonable choice for an optimal estimate all coincide.

CHAPTER III

SENSOR NETWORK TESTBED

The sensor network testbed comprising eight sensor nodes is used for tracking people walking in an aisle using webcams connected to individual sensor nodes. The objective of this testbed is to demonstrate the proposed approach for carrying out dynamic software reconfiguration on sensor networks.

The first section of this chapter describes the objective of the testbed. The second section describes the hardware infrastructure of the experimental testbed with details about the sensor devices and the network configuration. The third section describes the software infrastructure, which includes the operating system and various open source software programs.

3.1 Objective of the Experimental Testbed

The purpose of the experimental testbed is to demonstrate the validity of the proposed approach for carrying out dynamic software reconfiguration on sensor networks. This work is based on utilizing an application deployed on the sensor network that can be dynamically reconfigured to perform satisfactorily in a changing environment. The Aislemonitor application (described in Chapter 6), which tracks people walking in an aisle, is utilized for this purpose. The OpenBrick sensor devices (described in the next section) form the sensor network. They are deployed along a straight line in an aisle. Each OpenBrick device hosts the Aislemonitor application.

The reconfiguration approach is tested by shutting down one of the OpenBrick devices, thus requiring its neighbors to reconfigure their application components dynamically to predict the position of the people in the range of the affected node. This involves switching of some components, viz. the Estimator and the DataCollector (described in Chapter 6), executing on the neighboring OpenBrick devices. The process of choosing appropriate components to form a new configuration is very computation intensive and cannot be typically performed on sensor nodes. This task is therefore performed on a more powerful machine like a base-station. The whole reconfiguration cycle involves monitoring the health of the network, communication with other sensor devices and the base station, choosing new components, generating reconfiguration scripts, communicating these to the individual sensors and actually switching the components executing on the sensors with the ones selected in the new configuration.

I have implemented a software infrastructure to carry out all the mentioned tasks necessary for enabling dynamic software reconfiguration on our sensor network testbed. For the purpose of prototyping and validating the proposed software reconfiguration approach, this testbed adequately replicates real world sensor networks and the application for tracking people in the aisle provides adequate representation of real world sensor network applications.

3.2 Hardware Infrastructure

The sensor network testbed consists of eight OpenBrick-E wireless sensor devices and a base station [32]. The OpenBrick-E has a small form factor (220 x 165 x 42 mm) and weighs only about 1200 g. It includes three RJ45 LAN connectors and built in USB-

based 802.11b wireless LAN with a standard 2 dbi antenna. The default configuration has a fan-less 533 MHz x86 compatible VIA C3 processor and 256 MB SDRAM. Software can be installed on a Compact Flash or on an optional Hard Disk. The eight OpenBrick devices used in the testbed are equipped with 256MB of RAM and a 20GB Hard Disk each. In addition to WiFi support, each OpenBrick device includes three USB ports, PS/2 mouse and keyboard ports, a video port for the monitor, two Serial ports and a Parallel port. An OpenBrick-E device is displayed in Figure 7.

Figure 7 - OpenBrick – E Device

Figure 8 - Logitech QuickCam Pro 4000 webcam

Each Openbrick device is connected to a Logitech QuickCam Pro 4000 webcam (Figure 8). The QuickCam is connected to the OpenBrick device through its USB port and acts as the video capture device. The webcam supports images with resolution up to 640 x 480 pixels and includes an advanced VGA CCD sensor.

The base-station executes the Windows XP operating system. It is connected to one OpenBrick device through a wired 802.3 LAN connection. The base station is used to carry out computation intensive tasks in the reconfiguration process. It has an Intel Pentium 4 2.53GHz Processor with 1 GB of RAM, a RJ45 LAN Connector and 100GB of storage space.

The eight OpenBrick devices are configured to form a private ad hoc wireless network. Each device is assigned a static IP address in the private wireless network. The private network is maintained through the use of an ESSID. The network infrastructure also includes a LINKSYS EF3124 24 port 10/100 Ethernet switch. The switch is part of the private 'wired' LAN network, which connects the base station to the first OpenBrick device. Devices connected to the wired network are also assigned static IP addresses. The 24-port switch is also used in development activities. The base station is assigned a static IP and connected to the switch. One OpenBrick device is also connected to the switch. The base station communicates with all other OpenBrick devices through this wired connection. The detailed network configuration is displayed in Figure 9.

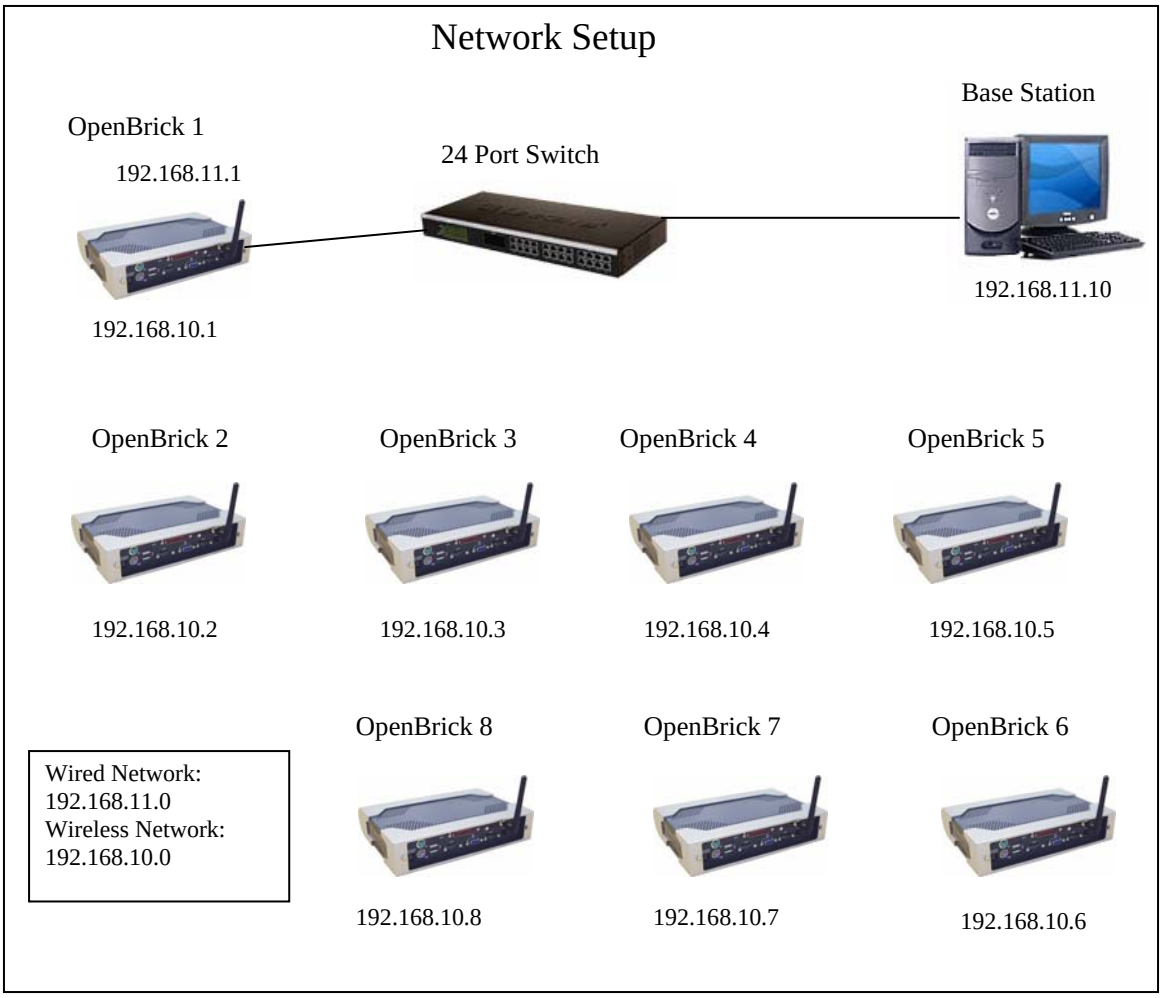

Figure 9 - Network Setup

Although the OpenBrick devices used in our experimental testbed are very powerful, equipped with a hard disk drive and powered by standard AC input, they can be easily adapted to use a flash card instead of the hard disk and a battery for the power supply. The device is portable and supports wireless communication. Any kind of sensor devices can be connected to OpenBrick devices through their USB, serial and parallel ports. The devices support Linux, which enables the use of all kinds of open source software for sensing and other activities. These added hardware facilities coupled with

the support for Linux make the OpenBrick device a very suitable match to test our software reconfiguration approach.

3.3 Software Infrastructure

The OpenBrick devices use Red Hat Linux 9 operating system with 2.4.20 kernel while the base station uses the Windows XP Professional operating system.

The communication between the OpenBrick devices occurs by exchanging data packets through datagram sockets using the User Datagram Protocol (UDP). The reconfiguration software and the applications generate UDP packets and route them to the destination using the Internet Protocol (IP). However, simple UDP sockets are not enough to carry out the communication over the entire wireless network. A node cannot send a message to another one outside its wireless range. This problem can be resolved by using a routing protocol that maintains route tables. The protocol refers to these tables and relays the packets through intermediate nodes before they reach the destination.

An implementation of the Ad hoc On Demand Distance Vector (AODV) [18] routing algorithm, a routing protocol for ad hoc mobile networks, provided by the National Institute of Standards and Technology (NIST) [33] is installed on each OpenBrick device. This enables the formation of a multi-hop ad hoc sensor network comprising of the eight OpenBrick devices. AODV is a method of routing messages between mobile computers. It allows these mobile computers, or nodes, to pass messages through their neighbors to nodes with which they cannot directly communicate. AODV does this by discovering the routes along which messages can be passed. AODV makes sure these routes do not contain loops and tries to find the shortest route possible. AODV

is also able to handle changes in routes and can create new routes if there is an error. The AODV program is configured to work only on the wireless network. This keeps the communication channel between the base station and the first OpenBrick device open through the wired 802.3 Ethernet connection.

An open source program for detecting motion is also installed on each OpenBrick device. This software, called Motion [34], captures frames from the webcams at a set frequency using Video for Linux (v4l) drivers. Subsequent images are then subtracted from each other and if the resultant frame contains non-zero values for some pixels, it indicates motion. Our application uses a modified version of this software where the frames captured every time cycle are compared to a background frame rather than the previous frame. The images captured by using the v4l drivers from the Logitec Quickcam webcam are in the YUV420P [35] format. For the purpose of detecting motion, the software just uses the Y or the Brightness value of the image. Y ranges from 16 to 235 or full brightness.

CHAPTER IV

RECONFIGURATION ARCHITECTURE

Sensor networks, typically deployed in inaccessible environments, are vulnerable to failure due to the dynamic nature of their target environments and their own resource limitations. However, the composition of these networks, which includes numerous collaborating sensors nodes, can be used to address these problems. If some sensor nodes become disabled, their neighboring nodes can be made to take up their responsibilities. Similarly, if too many nodes become redundant, some of them can be suspended, thus adding to the saving of precious resources. This dynamic behavior of the sensor networks can be achieved by reconfiguring the software components executing on each individual sensors.

This chapter describes the proposed approach for performing dynamic software reconfiguration in sensor networks. The first section of this chapter describes the problem of software reconfiguration. The second section states the approach in detail for carrying out software reconfiguration in sensor networks. The third section defines the scope of the reconfiguration architecture.

4.1 Software Reconfiguration Problem

Sensor networks are often required to operate in inaccessible and dynamic environments that impose varying functional and performance requirements. This accentuates the need for software systems that can adapt to new conditions by reconfiguring themselves by detecting internal and external changes to the system and

reflecting on the event occurrences. Ad hoc wireless sensor networks, in particular, must be designed with adaptation capabilities that enable them to handle a multitude of operating conditions. However, reconfiguration in such systems presents significant challenges because of the severe constraints in energy, computation, and communication resources. Computation intensive processes like choosing new configurations of components from a large design space cannot be performed on the individual sensors. Runtime technologies that allow software to evolve as system requirements and/or its environment change are critical to the development and deployment of such systems [16].

4.2 Proposed Solution

This thesis presents an approach for building self-adaptive sensor networks based on Model-Integrated Computing [37]. This approach utilizes explicit models of the *design space* of the embedded application. The design space is captured by formally modeling all the software components and their interactions that together constitute an application. The modeling of the application is based on the Asynchronous Data Flow model of computation [28]. Components interact with each other only when exchanging information through input and output ports. System requirements are expressed as formal constraints on operational parameters such as power consumption, latency, accuracy, and other QoS properties that are measured at runtime. These constraints are expressed in the Object Constraint Language (OCL) [38]. The constraints and models are embedded in the running application forming the *operation space* of the system. Reconfiguration is the process of transitioning from one point of the operation space to another [16].

Finding the components to be included in a new configuration can be considered a search problem in the operation space. The exploration of the operation space is a challenging problem since it must be performed within stringent time bounds and resource constraints. An efficient approach for performing this search is based on (1) parameterized constraints captured in the embedded models, and (2) online constraint solving using a combination of symbolic constraint satisfaction and linear programming. Once a new configuration that satisfies all the constraints is found, the reconfiguration can be accomplished by online software synthesis targeting either an interpreted language or a command interface.

Reconfiguration thus involves two major tasks – (1) finding the new configuration and (2) switching or reconfiguring the components that are actually executing on the individual sensors. The first task is performed by a controller that runs on the base station while the second task is performed on individual sensors by specialized switching components. The reconfiguration architecture is displayed in Figure 10.

During design time, the entire application is modeled in the Generic Modeling Environment (GME). The Sensor Network Reconfigurable Applications Modeling Language (SNRAMoLa) for component based sensor network applications supports the modeling of alternate implementations of the same components and explicit representation of constraints in OCL. A series of critical QoS parameters are also modeled as attributes to individual components. Our models include the ‘power’ attribute. The constraints are resolved over these attributes by the DEsign Space ExploRation Tool (DESERT) and valid configurations of the application are generated. SNRAMoLa is described in detail in Chapter 5.

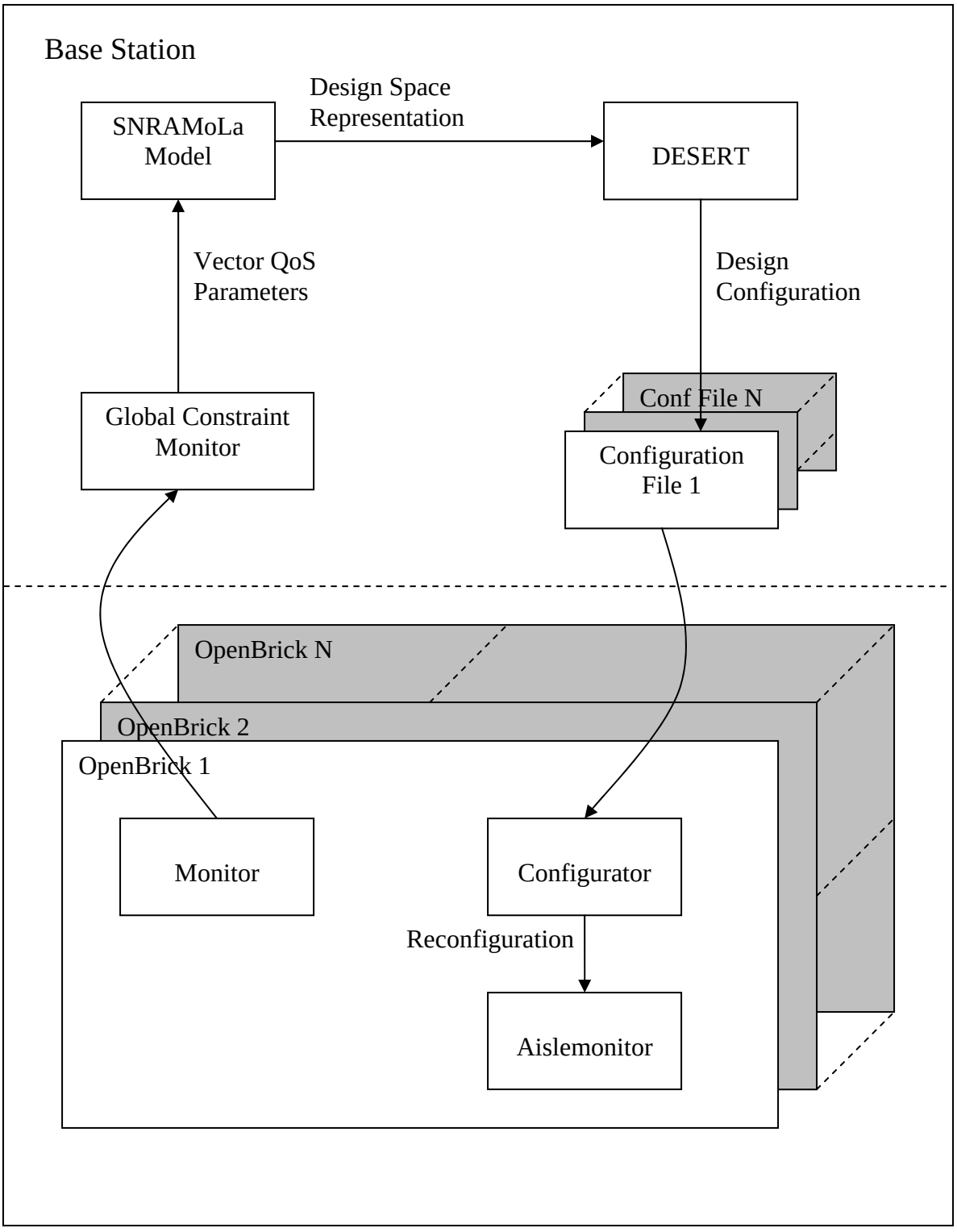

Figure 10 - Reconfiguration Architecture

During run-time, upon detecting failure, the Global Constraint Monitor (GCM) (located on the base station) updates the values of the critical QoS attributes in the application models. A change in the values of the attributes invokes the reconfiguration process.

During reconfiguration, the first task of finding a new configuration is performed by a controller program executing on the base station. The SNRAMoLa models of the application, which include all application components and associated constraints, form the design space for selecting the valid configurations of the application. A valid configuration includes all those components from the SNRAMoLa models that satisfy all the constraints. The design space exploration tool, DESERT [36] (described in Chapter 2), evaluates the constraints and selects an appropriate configuration by pruning the design space. DESERT performs this task by applying these constraints on specific QoS attributes associated with the components. In our case, DESERT prunes the design space based on the value of the power attribute in each component. This power attribute reflects the power available on each node. The output of DESERT is used in the generation of a set of configuration files, one for each sensor. The reconfiguration process then involves sending the new configuration files to the individual OpenBrick devices over the ad hoc wireless network.

The second task is performed by the Configurator component executing on individual OpenBrick devices. It reads from the new configuration file sent by the base station and, stops, rewires or starts active or dormant application components already present on the sensor nodes. In addition to the Configurator, the Monitor components executing on each device, monitor the health of the network. These communicate with

the GCM component executing on the base station in case of failure in the network. The GCM in turn updates the critical attributes like power in the application model, which invokes the reconfiguration process. The Configurator, Monitor and GCM are described in Chatper 7.

4.3 Main Contribution

The main contributions of this thesis include:

1. A modeling environment called SNRAMoLa developed in GME used for modeling component based sensor network applications.
2. An interpreter that converts SNRAMoLa models to DESERT input.
3. An interpreter that generates configuration files using SNRAMoLa models and DESERT output.
4. A software reconfiguration agent called Configurator that switches components executing on the sensor nodes.
5. A program called Packet Forwarder that relays messages between the sensor nodes and the base station.
6. Monitoring components that monitor the health of the sensor network and communicate node failures to the base-station.
7. A program called the Global Constraint Monitor (GCM) that updates the QoS parameters in the SNRAMoLa models upon receipt of node failure messages from the Monitors on the sensor nodes.
8. The Aislemonitor application for tracking people walking in an aisle. The application is used for evaluating the software reconfiguration architecture.

CHAPTER V

MODELING RECONFIGURABLE SENSOR NETWORK APPLICATIONS

The representation of multiple software components that perform various tasks in an application along with their interactions poses a significant challenge to software engineers. Such a representation is vital for carrying out software reconfiguration to understand the current status of the application executing on multiple sensor nodes. Textual representation of such applications is error-prone and of little use in managing the complexity of the application. Even in the simplest scenario a more expressive representation of application components and the interconnections between them using modeling tools can help users avoid errors and help others understand the application. With more complex components this becomes an absolute requirement. Model Integrated Computing [37] in general and the Generic Modeling Environment (GME) in particular can meet these challenges [16].

This chapter presents a modeling paradigm for representing reconfigurable sensor network applications in the Generic Modeling Environment. The first section of the chapter describes this modeling paradigm in detail. The second section describes the modeling of a simple sensor network configuration using the Aislemonitor application described in Chapter 6. The third section describes the modeling of an application with multiple configurations. It also includes example models of the Aislemonitor application described in Chapter 6.

5.1 Sensor Network Reconfigurable Applications Modeling Language (SNRAMoLa)

The paradigm for modeling component based reconfigurable software systems has been developed in the Generic Modeling Environment (GME). Figure 11 shows the meta-model for the Sensor Network Reconfigurable Applications Modeling Language (SNRAMoLa). This modeling paradigm is based on the Asynchronous Data Flow (ASDF) model of computation [28].

Component based software systems are comprised of a number of software components interacting with each other. Such applications can be intuitively modeled as graphs. Individual components are represented as nodes while their interactions as arcs in the graph. Two nodes in the graph are connected if they interact with each other. The components are connected through input-output objects called ports.

ASDF is a special case of data flow, a hardware and software methodology popular for representing parallel computation. Under the data flow paradigm, algorithms are described as directed graphs where the nodes represent computations (or functions) and the arcs represent data paths. The data flow principle is that any node can *fire* (perform its computation) whenever input data is available on its incoming arcs. A node with no input arcs may fire at any time. This implies that many nodes may fire simultaneously, and hence also represents concurrency.

In this thesis, Component based sensor network applications are designed as a group of independent processes communicating with each other using inter-process communication directives and constructs. This kind of software architecture enables the dynamic switching of components executing in the application. Such applications can be easily represented in the ASDF model of computation. The use of ASDF allows the

mapping of the sensor network application onto an ASDF graph such that each node in the graph represents a process and the arcs connecting two nodes represent the inter-process communication. The processes in the application, like the nodes in the ASDF model, are also data driven and fire only when they receive data from another process. The processes that are not connected to any other process may fire at any time as in the ASDF model.

SNRAMoLa enables the user to represent sensor network applications in the form of a graph. This application graph is composed of components that exchange data through ports. The core concepts in the modeling language are *Component-s, InPort-s, OutPort-s, DataFlow connections, Choice-s* and *Condition-s* (described later in this section). The *Sensor* and *SensorFolder* objects contain the application graph, which is composed of the core objects. The *ComponentsFolder* object contains all the non-reconfigurable Component objects. Non-reconfigurable Component objects cannot be replaced by any other components in the application by DESERT during the reconfiguration process and are always included in the application configuration.

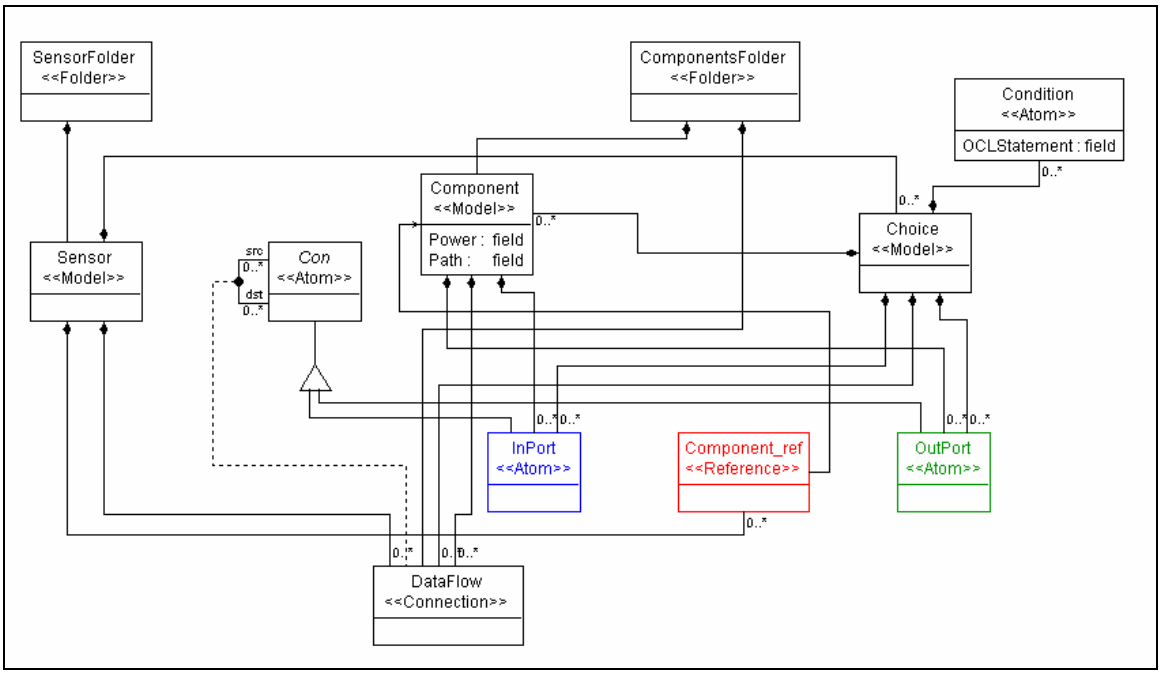

Figure 11 - SNRAMoLa Meta Model

5.1.1 SensorFolder

Each SNRAMoLa application model contains exactly one SensorFolder object. The SensorFolder acts as a container for all the Sensor objects, which model sensor devices, in our case, OpenBrick devices.

5.1.2 ComponentsFolder

Each SNRAMoLa application model contains exactly one ComponentsFolder object. This object contains all the non-reconfigurable Component objects that are included in the application. The components are then only referred in the actual application graph built in the Sensor objects. Typically, in a sensor network application, most sensors will have the same application components executing on them and not all components will need to be replaced during reconfiguration. Many of the components

included in the application configuration will be non-reconfigurable. This modeling feature removes the duplicate declaration of common components, which are always included in all the configurations on each Sensor.

5.1.3 Sensor

Separate Sensor objects for each sensor are declared inside the SensorFolder object. Sensor objects represent the actual sensor nodes in the sensor network. In our case, each Sensor object represents an OpenBrick device. This modeling feature enables the user to model different applications (applications composed of different components) for different sensors in the network. The graphs for the application executing on each sensor are then built inside these Sensor objects. The application graph is composed of the core Component, Choice and Condition objects.

5.1.4 Component

A Component object in SNRAMoLa represents a separate process in the application executing on the sensor devices. Components may contain InPort(s) and OutPort(s) objects if they exchange data with other Components in the application. These ports are used to pass messages between components. A Component is reconfigurable if it can be replaced by another Component in the application graph during the reconfiguration process and is non-reconfigurable if it is irreplaceable in the application graph. Non-reconfigurable components are declared in the ComponentsFolder folder object and referenced in the application graph built on individual Sensor objects while reconfigurable components are declared in the Choice objects, which form containers for

alternative Components. Each Component object also has an attribute called ‘Path’ which identifies the physical path of the executable that is invoked when starting the Component. This attribute is used by the Configurator component on the individual OpenBrick devices to execute the corresponding process represented by that Component.

5.1.5 InPort

An InPort object in SNRAMoLa represents an input port of a Component object. It is declared inside a Component object and used to accept data from another Component object. This functionality is implemented in the reconfiguration software infrastructure using shared memory. A Component object can have any number of InPort objects but each InPort object can be connected to at most one corresponding OutPort object declared inside another Component or Choice object using the DataFlow connection object.

5.1.6 OutPort

An Outport object in SNRAMoLa represents an output port of a Component object. It is also declared inside a Component or Choice object and used to send data to another Component object. This functionality is implemented by the reconfiguration software infrastructure using shared memory. A component can have any number of OutPort objects but each OutPort object can be connected to at most one corresponding InPort object, declared inside another Component object, using the DataFlow connection object.

5.1.7 DataFlow

A DataFlow object is a connection object that links an output port of a Component object represented by an OutPort object with an input port another Component object represented by an InPort object. It models the Asynchronous flow of data from one application component to another. The DataFlow object along with the InPort and OutPort objects is implemented by reconfiguration software infrastructure using shared memory.

5.1.8 Choice

As the name suggests, a Choice object facilitates the user to model reconfigurable or mutually replaceable Component objects in the application graph. At any given instance, only one process from the collection of processes represented by the Component objects declared in a given Choice object actually executes in an application. A Choice object also contains a Condition object, which specifies the condition expressed in OCL in its Expression attribute. During the reconfiguration process, DESERT evaluates all the constraints modeled as Condition objects over all the Component objects declared in the respective Choice objects and selects only one Component object to be included in the final application graph from each Choice object. DESERT does this selection based on the value of the Power attribute of the Component objects.

5.1.9 Condition

Each Choice object contains at least one Condition object that specifies a constraint expressed in OCL. During the reconfiguration process, this constraint is

evaluated by DESERT over the collection of reconfigurable Component objects also contained in the same Choice object and only those Component objects that satisfy the constraints are selected for inclusion in the final application graph. The Condition object has an attribute called Expression, which is used to express the constraint in OCL.

The SNRAMoLa paradigm enables the user to model complex component based sensor network applications in a more intuitive manner. The communication links between various components are clearly expressed using the InPort, OutPort and DataFlow objects. The paradigm enables the user to model components that can be replaced by others during runtime along with the constraints that govern the selection of the appropriate components from the collection of alternatives. Using SNRAMoLa, the user can visualize the applications executing on individual sensor nodes along with all the active and passive components of the application and maintain different versions of the application on individual sensors if needed.

5.2 Modeling of Single Configuration

An application with only one valid configuration can be modeled in SNRAMoLa using only Component(s) and DataFlow objects. Such a model does not contain any Choice and reconfigurable Component objects. The application graph of such an application contains all the Component objects declared in the model. The interactions between Components are modeled as DataFlow objects connecting their OutPort objects with their InPort object. All the Component objects are declared in the ComponentsFolder and referred to in the application graph built in the Sensor objects. In single configuration applications, there is no need for reconfiguration as only non-reconfigurable components are actually used in the application.

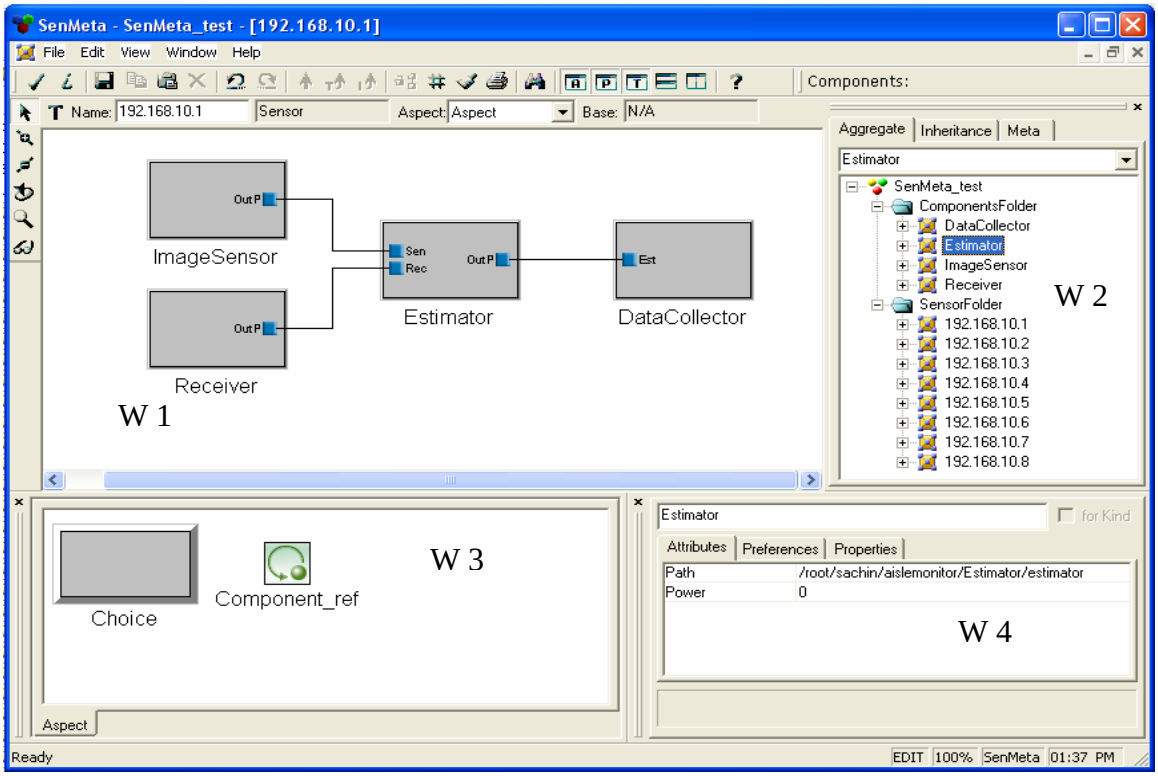

Figure 12 - SNRAMoLa Model of Aislemonitor with Single Configuration

Figure 12 shows the SNRAMoLa model of the Aislemonitor (described in chapter 6) application with a single configuration. As shown in the figure, the application graph built in a Sensor object displayed in Window 1 (W1) consists of four Component objects - Receiver, ImageSensor, Estimator and DataCollector. The Receiver and the ImageSensor send data to the Estimator and the Estimator sends data to the DataCollector through their respective OutPort objects (described in Chapter 6). The attributes of the Estimator Component are shown in W4. The 'Path' attribute of the Estimator specifies the physical location of the executable of the process represented by the Estimator Component on the OpenBrick device. W2 displays all the modeled objects. The ComponentsFolder object contains all the non-reconfigurable components, in this case, all the three components. The SensorFolder object contains all the Sensor objects identified by their IP addresses in the network. The components declared in the ComponentsFolder object are referenced in the application graph, displayed in W1. W3 displays the modeling components that can be defined in W1. In this case, Choice and Component_reference objects can be defined on the Sensor object, but since this a single configuration application, only Component_reference objects are defined. The InPort and OutPort objects are declared as shown in Figure 13.

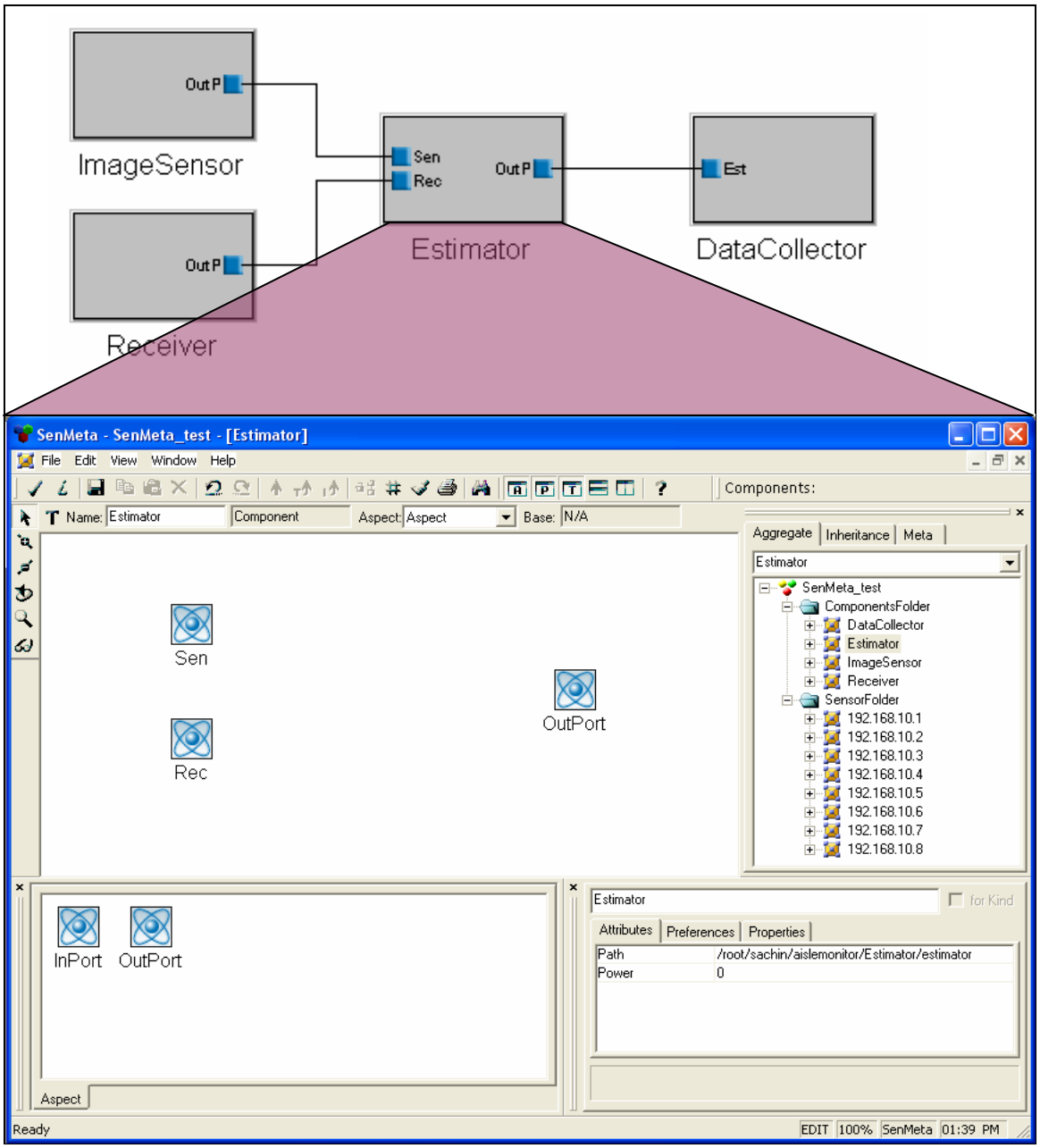

Figure 13 - Declaration of InPort and OutPort Objects in Estimator Component

5.3 Modeling of Multiple Configurations

Applications with multiple configurations can be modeled in SNRAMoLa using Choice, Condition and reconfigurable Component objects. Each Choice object contains Component objects that can be replaced by other Components declared in it. The Condition object, also present in the Choice object, governs the selection of the proper Component based on the constraints expressed in its 'Expression' attribute. The rest of the model is similar to that of an application with a single configuration. Reconfigurable components are encapsulated into Choice objects, which are included in the application graph.

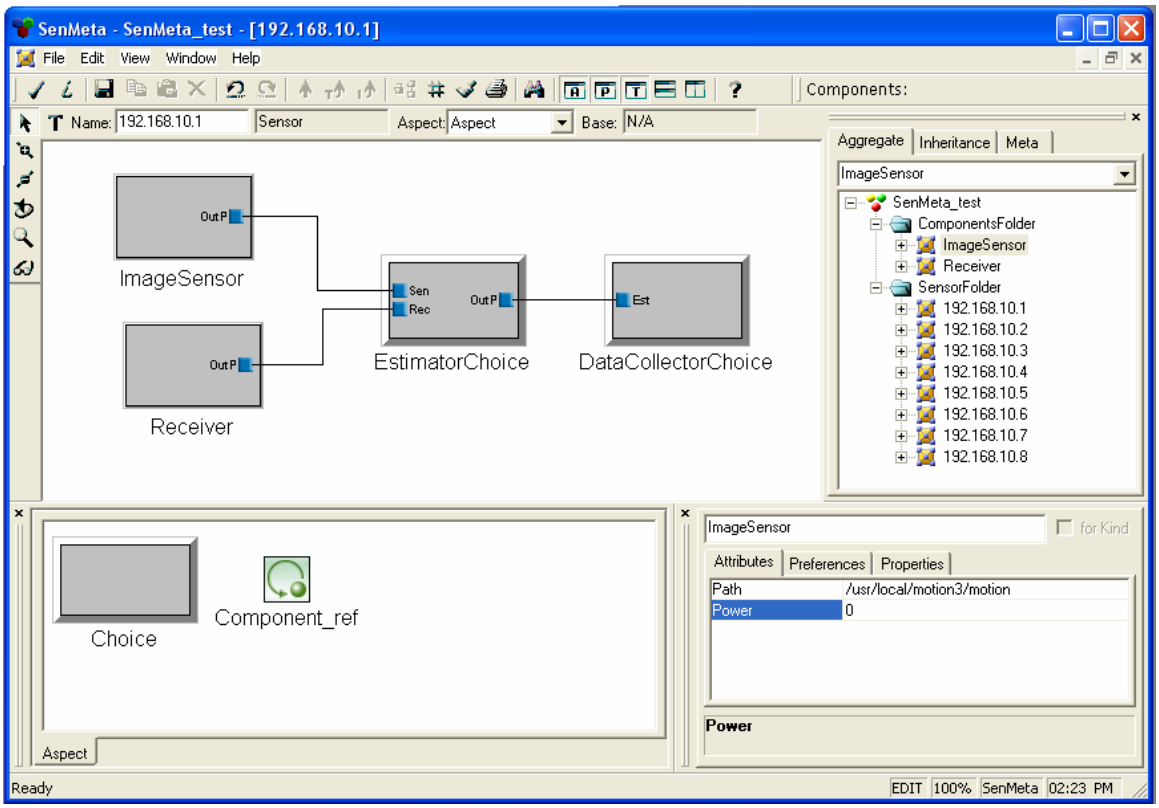

Figure 14 - SNRAMoLa Model of Aislemonitor with Multiple Configurations

Figure 14 shows the SNRAMoLa model of the Aislemonitor application. It shows Choice objects EstimatorChoice and DataCollectorChoice in place of the Estimator and DataCollector Component objects as shown in the model for a single configuration.

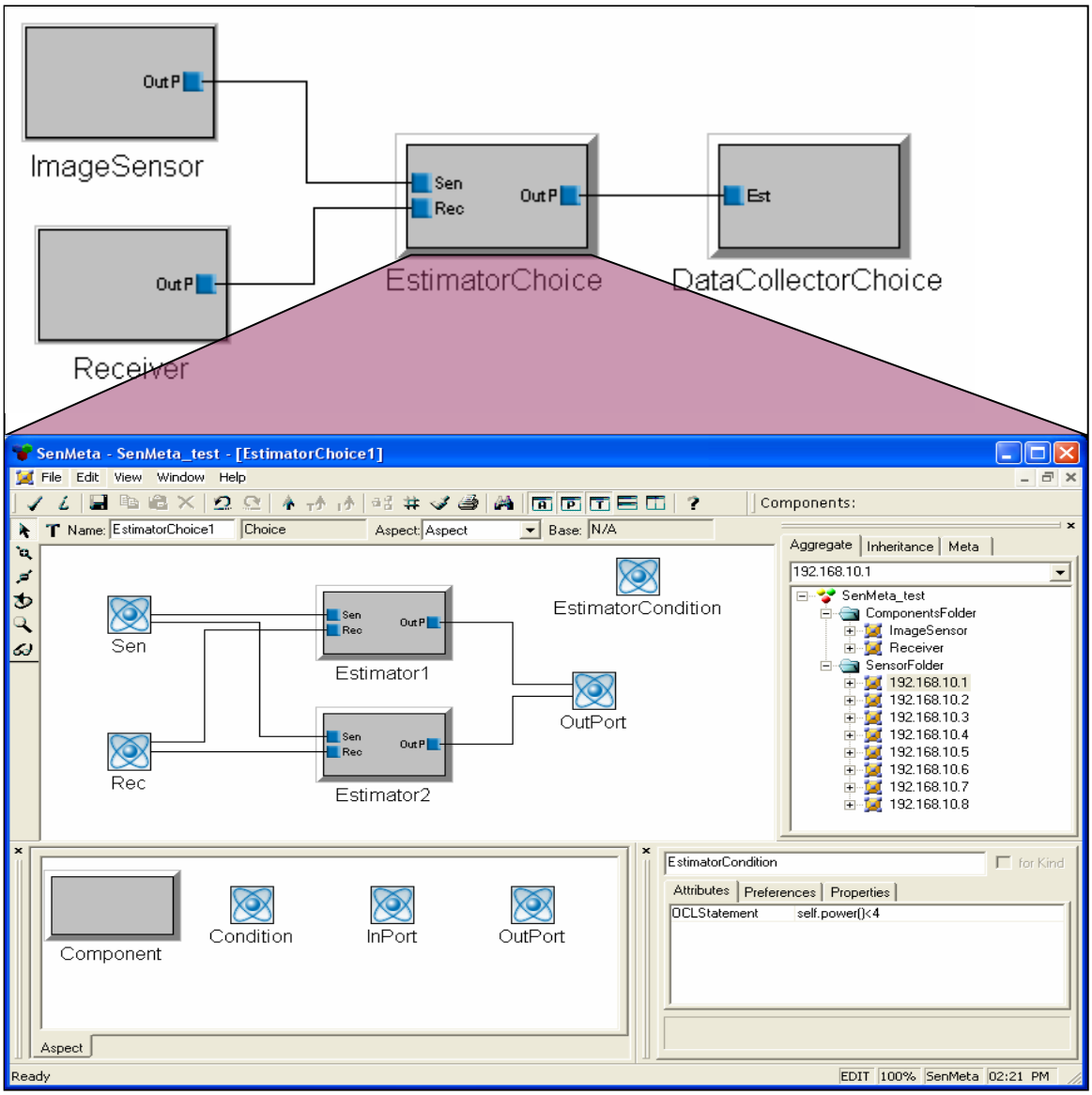

Figure 15 - Estimator1, Estimator2 and EstimatorCondition in EstimatorChoice Object

The EstimatorChoice and DataCollectorChoice objects contain Estimator1 and Estimator2 and DataCollector1, DataCollector2 Components, respectively, along with the Condition objects EstimatorCondition and DataCollectorCondition as shown in Figure 15.

The OCL expression in the EstimatorCondition object, “self.Power()<4”, is evaluated by DESERT during the reconfiguration process and the Component with the ‘Power’ property value less than 4 is chosen in the returned configuration. Thus in this example Estimator 1 will be selected as its Power attribute value is 0. The attribute value for Estimator 2 is 5. The choice of the Component objects to be included in the final application graph can be governed by changing their ‘Power’ attribute values, which is what the Global Constraint Monitor does.

5.4 Component Based Applications Architecture

The previous sections presented a modeling paradigm for representing reconfigurable sensor network applications in the Generic Modeling Environment. It utilizes the modeling constructs of Components and DataFlow connections to accurately represent the components of an application along with their interactions. However, just the representation of applications in user friendly models is of little consequence if these models are not used to generate useful output. SNRAMoLa models are used to generate configuration scripts. A configuration script indicates which components are actually included in an application and which are not. It also includes the information about the interaction between the included components. The software engineering architecture enables the dynamic switching of processes according to the configuration script.

The proposed approach suggests that software reconfiguration for adaptive software systems should be a simple switch in the executing components. However, to achieve this switch, one should be able to turn the required components ‘off’ and turn their replacements ‘on’ during runtime. This can be achieved if the components do not belong to the same compiled executable. This led to the implementation of each component as a separate process.

Having mapped a component to a process, the next step was to build a software reconfiguration architecture (explained in Chapter 7) to manipulate these processes. SNRAMoLa models of the application represent inter-component communication through input and output ports and DataFlow connections. These connections can be implemented using inter-process communication constructs. Shared Memory is an efficient means of passing data between programs. The software reconfiguration architecture creates a memory block for each connection, and the source and the destination processes just connect to it. The source and destination processes use pointers to access the shared memory to read and write data. The names of these pointers directly map to the names of the input and output ports in the SNRAMoLa models. After considering the use of pipes, signals, and semaphores in the implementation, shared memory was chosen as it was efficient, easier to manage and fit well with the modeling language in the reconfiguration architecture.

Though the present implementation assumes offline installation of all the components on each sensor node, it can be easily extended to accept installation of new processes online, without affecting the executing application, and then reconfiguring the application to include the new process.

CHAPTER VI

AISLEMONITOR SENSOR NETWORK APPLICATION

Aislemonitor is a sensor network application developed to perform one-dimensional tracking of people walking in an aisle. This application is deployed on the sensor network testbed described in Chapter 3 and is used for evaluating our software reconfiguration approach. A copy of the application is installed on each OpenBrick devices. The application is composed of components for motion detection, tracking, data recording, and communication. There are two implementations of the tracking and data recording components. Though all the components are installed on each OpenBrick device, not all of them execute at the same time. The components that execute together form a configuration of the application.

One such configuration of the application includes components that just track the people within the range of individual OpenBrick devices. Another configuration of the application includes components that, in addition to tracking people within range, can also predict the position of the people in the range of the neighboring OpenBrick device. A change in the network configuration caused by disabling one OpenBrick device creates a gap in the sensing range of the network. This gap can be filled by switching the components executing on the neighboring OpenBrick device by those that can predict the position of the people in the range of the disabled device. The functioning of the proposed reconfiguration infrastructure is demonstrated by dynamically switching components of the Aislemonitor application upon detecting failure in the neighboring

OpenBrick device. The new configuration of the application facilitates the tracking of people within range of the disabled device.

The first section of this chapter describes the configuration of the Aislemonitor application which includes components that track people in the range of only their host OpenBrick devices. The second section describes the configuration that includes components, which in addition to tracking people in their own range, can also track people in the range of neighboring OpenBrick devices. The third section describes all the components of the Aislemonitor application.

6.1 Aislemonitor Application – Configuration 1

Configuration 1 of the Aislemonitor application includes only those components that can track people within the range of their host OpenBrick devices. This configuration is executed on all the OpenBrick devices whose right neighbors are active. Figure 16 shows the operational setup for this application. The sensor network comprising the eight OpenBrick devices is deployed in the aisle as shown in the Figure 16. The collective range of the sensor network is 37 feet. The OpenBrick devices are kept equidistant from each other along a straight line in the aisle so that the ranges of their webcams overlap.

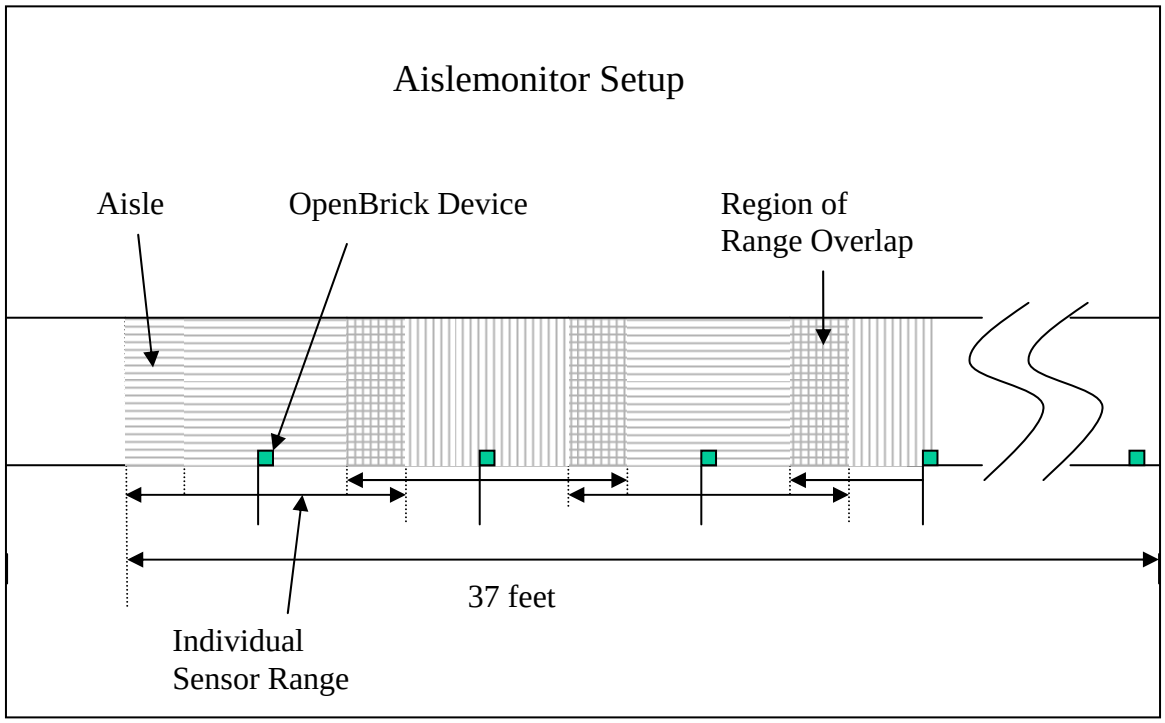

Figure 16 - Aislemonitor Setup

The application tracks people walking in an aisle by using the webcams and estimating their position in the aisle. When a person moves from the range of one device

to another, or in the range of overlap (area within the range of two consecutive webcams), the device communicates with its neighbor and hands over the tracking of that person to the neighbor. The components included in this configuration of Aislemonitor are Receiver, ImageSensor, Estimator1 and DataCollector1 (described in section 6.3).

6.2 Aislemonitor Application – Configuration 2

The second configuration of the Aislemonitor application is executed on the OpenBrick device when its right neighbor shuts down. The failure of one OpenBrick device causes a hole in the sensing range of the network. The components of the neighboring OpenBrick devices then take over the prediction of the people in this range by predicting their position.

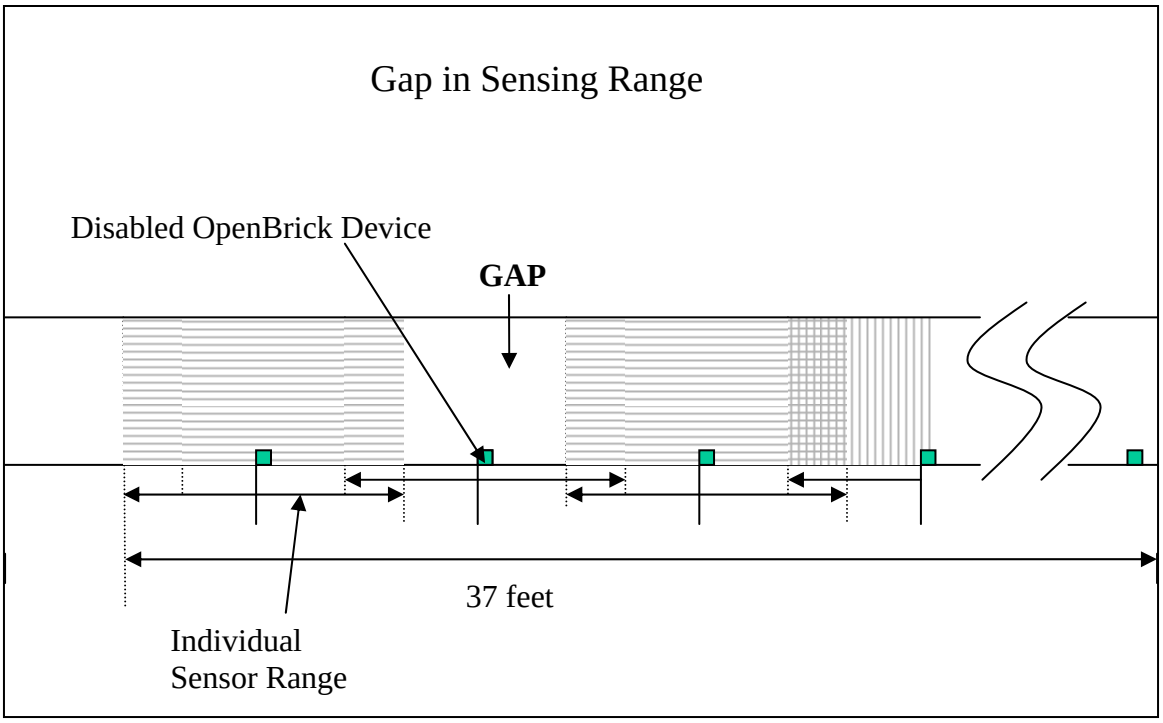

Figure 17 Aislemonitor Configuration 2 – Gap in the range

Figure 17 displays the hole caused in the sensing range of the network. When the second OpenBrick device in the network shuts down, a gap is created between the first and the third OpenBrick devices. This gap is reduced by reconfiguring the application executing on the first OpenBrick device. The new configuration includes the same Receiver and ImageSensor components also included in the first configuration. However, during reconfiguration, the Estimator1 and DataCollector1 components are dynamically replaced by the Estimator2 and DataCollector2 (explained in section 6.3) components. The Receiver and the ImageSensor components continue to execute even during reconfiguration. Only the Estimator1 and DataCollector1 components are stopped and their alternatives are started. The second configuration of the Aislemonitor is able to predict the position of people in the range of a disabled OpenBrick device if that device is the right neighbor of the host device.

6.3 Components of the Aislemonitor Application

The Aislemonitor application is composed of six components called *ImageSensor, Receiver, Estimator1, Esitmator2, DataCollector and DataCollector2,* which communicate with each other through shared memory to exchange data. Estimator1 and Estimator2 are reconfigurable components and only one of them executes on the OpenBrick device at any given time. The same holds true for the DataCollector1 or DataCollect2 components. The SNRAMoLa model of this application is shown in Chapter 5. Though all the six components are installed on all the OpenBrick devices, at a time, only four of them execute on the devices. Estimator1 and DataCollector1 execute together and Estimator2 and DataCollector2 execute together. Figure 18 shows the

functional graph of the Aislemonitor application. As shown in the figure, data flows from the Receiver and ImageSensor components to the Estimator1 or Estimator2 components and from there to the DataCollector1 or DataCollector2 components, respectively, through the specified ports.

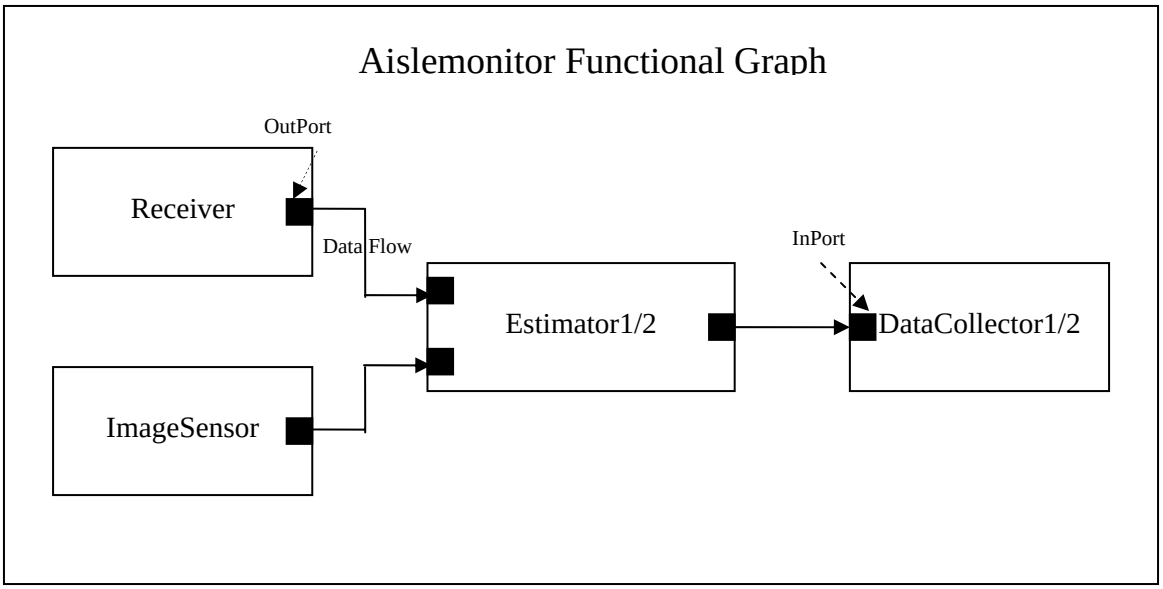

Figure 18 - Aislemonitor Functional Graph

6.3.1 ImageSensor

The ImageSensor component is an extension of the motion detection software – Motion [34]. Motion compares images captured using the webcam with a background image of the aisle at a set frequency and raises an event for 'motion detected' if the difference between the two images is more than the set threshold. The ImageSensor component then calculates the center of mass of the person in the image. The component then sends this value to the Estimator component, which uses a Kalman filter to estimate the correct position on the person in the aisle. The ImageSensor only works in situations where there is only one person in the field view of its webcam at any given time. The

Aislemonitor application assumes that only one person can be in the range of a particular webcam at any given time. There could be more people in the range of the whole network at the same time, but not within range of the same OpenBrick device.

6.3.1.1 Calibration

Before deploying the application, each webcam is calibrated offline. This involves the mapping of each pixel in the image onto the actual aisle. As we are only interested in the position of the person in the aisle in the horizontal direction and not in his/her height, we only map the pixel at the x coordinate of the center of mass of the person in the image to the aisle. If *Mx* is the x coordinate of the center of mass in the image (in pixels), then the real x coordinate of the person in the aisle, *Rx* (in inches) is:

```
Rx = Mx * (right_range_limit – left_range_limit) / image_width;
```

The variables right_range_limit and left_range_limit are the ranges of the webcams in the right and left directions respectively. Their units are in inches. The width of the image is specified in image_width and its unit is pixels. We have set the width of the image to 160 pixels and the right_range_limit and left_range_limit vary from camera to camera. The unit of the real x coordinate *Rx* is inches. For example, if the ImageSensor calculates the x coordinate of the center of mass as 15, and the left range of the webcam is 0 inches, while the right range is 75 inches, then the position of the person in the aisle can be calculated as *Rx* = 15 * (75 – 0) / 160 = 7.03 inches.

6.3.1.2 Initialization

During initialization, the ImageSensor starts the webcam and stores a background image of the aisle in the variable background_image.

```
unsigned char *background_image;
background_image=malloc(image_size);

//Initialize video capture device – webcam
device = vid_start(image_width,image_height);

//Capture background image
background_image=vid_next(image_width,image_height);
```

All images are captured as unsigned character array frames using Video for Linux (v4l) drivers. After initialization, the ImageSensor goes into an infinite loop and performs the activities of (1) capturing new images, (2) calculating center of mass if motion is detected, (3) performing the calibration mapping and (4) passing the value returned to the Estimator at the set frequency.

6.3.1.3 Image Difference

After initialization, the ImageSensor captures images from the webcam at a set frequency. We have set the frequency at four frames per second. This value was chosen to minimize the energy lost in computations during iterations while maintaining the tracking capability of the application. During every iteration, the ImageSensor subtracts the new image from the background image, which is stored in the variable background_image during initialization. The result obtained is stored in the variable difference_image.

```
difference_Image = |background_image – new_image|;
```

The images obtained from the webcams are in the YUV420P format. For the purpose of detecting motion, only the Y values or brightness of the image are taken into consideration. The subtraction of the new image from the background image causes only the differences between the two images to be highlighted. The difference image displayed in Figure 19 highlights the pixels that are different from those in the background image. The image clearly shows a person walking in the aisle.

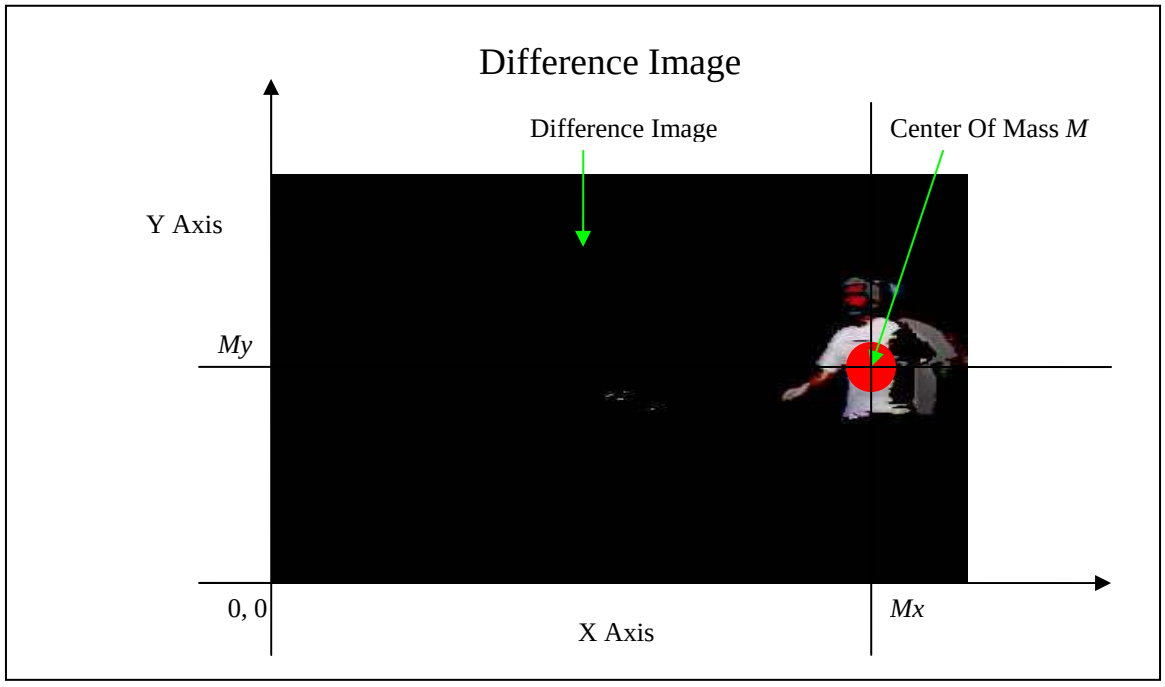

Figure 19 - Difference Image

6.3.1.4 Center of Mass Calculation

If the number of pixels that have changed in the new image is greater than the set threshold, the ImageSensor generates an event for 'motion detected' and calculates the center of mass of the image in terms of brightness.

$$Mx = \frac{\sum_{x=1}^{imagewidth} X \sum_{y=1}^{imageheight} M(x,y)}{\sum_{x=1}^{imagewidth} \sum_{y=1}^{imageheight} M(x,y)};$$

Mx is the x coordinate of the center of mass *M* in the image and *M*(*x*,*y*) is the intensity of the pixel at coordinates (x, y) in the difference image.

6.3.1.5 Calibration mapping

The ImageSensor then maps the x coordinate of the center of mass *Mx* from the difference image to the actual location on the aisle. The value *Rx* returned after the mapping is the relative position of the person in the aisle in inches from the left range limit of the webcam.

```
Rx = Mx * (right_range_limit – left_range_limit) / image_width;
```

6.3.1.6 Communication with Estimator

The ImageSensor component has one output port called 'OutPort'. This port is connected to the input port called 'Sen' of the Estimator component (shown in Chapter 5, Figure 12). After the mapping of the center of mass in the image to the actual image, the

value returned is passed to the Estimator component using shared memory where it is used by the Kalman Filter [30] to estimate the correct position of the person in the aisle.

6.3.2 Receiver

The Receiver component receives data packets from neighboring OpenBrick devices. If the data contains the position of the person entering the range of the host OpenBrick device, the Receiver passes on the data to the Estimator component through its ‘OutPort’ output port. The output port of the Receiver is connected to the ‘Rec’ input port of the Estimator by shared memory (Chapter 5, Figure 12).

6.3.2.1 Initialization

During initialization, the Receiver component opens up a port which listens for messages coming from the neighboring OpenBrick devices. The Receiver component then goes into an infinite loop to continuously listen for new incoming messages. When it receives a new message, it interprets it and then goes back to listening.

6.3.2.2 Runtime Operation

When a person enters the range of overlap between two OpenBrick devices, the device whose range the person is about to exit, sends a message to its neighbor in whose range the person is about to enter. This message contains the position and speed (explained in the next section) of the person. When the Receiver component receives this data from its neighbors, it passes the data to the Estimator (Estimator1 or Estimator 2) component through its output port. This data is then used by the Estimator to initialize the Kalman Filter (explained in the next section).

6.3.3 Estimator1

The Estimator1 (referred to as Estimator in this section) component implements a Kalman Filter [30], which takes the value passed by the ImageSensor component as input to calculate the position of the person in the aisle. The Estimator component has two input ports called ‘Rec’ and ‘Sen’. These ports are connected to the Receiver and ImageSensor components respectively. The Kalman Filter implemented in this component acts on the data passed by the Receiver and ImageSensor components.

6.3.3.1 Initialization of Kalman Filter

During runtime, the Kalman Filter is initialized every time a neighboring OpenBrick device hands over the tracking of a person exiting from its range and entering the range of the host OpenBrick or when a new person walks into the range of the host OpenBrick and the Kalman Filter has not already been initialized. In the first case, the Estimator1 uses the values of position and speed passed by the Receiver component to initialize the Kalman Filter, while in the second case, the Estimator uses the position value passed by the ImageSensor and a constant for the speed in the initialization of the Kalman Filter. The default value for speed was measured and set at 16 inches per second.

```
if(Data received from ImageSensor)
{
      //Constant = 16 inches/second.
      Kalman_Initialize(position,Constant);
}
else if(Data received from Receiver)
{
      Kalman_Initialize(position-left_range_limit,speed);
      //position indicates actual position in aisle relative to
      //start of aisle.
      //position and speed obtained from neighboring OpenBrick through
      //Receiver component.
}
```

6.3.3.2 Using the Kalman Filter

We consider the system dynamic equations for one dimensional tracking in the aisle:

$$x_{n+1} = x_n + \dot{x}_n T$$
$$\dot{x}_{n+1} = \dot{x}_n$$

In reality, the person will not move with a constant speed for all the time as indicated by the above equations. To model the uncertainty in the person's speed, the equations are modified with the addition of a random noise u_n to the person's speed. This gives rise to the following stochastic model:

$$x_{n+1} = x_n + \dot{x}_n T$$
$$\dot{x}_{n+1} = \dot{x}_n + u_n$$

The observation equation links the actual data x_n to the measured data y_n:

$y_n = x_n + v_n$, where v_n is the measurement noise.

The same equations in matrix form are expressed as follows:

$$X_{n+1} = \Theta X_n + U_n$$

where X_n is the state vector, Θ is the state transition matrix and U_n is the system noise vector.

The observation equation in matrix form is:

$$Y_n = MX_n + V_n$$

where Y_n is the measurement vector, M is the observation matrix and V_n is the observation noise vector.

The system dynamic equations of the stochastic model along with the observation equation give rise to the following matrices:

$$X_n = \begin{bmatrix} x_n \\ \dot{x}_n \end{bmatrix},\ \Theta = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix},\ U_n = \begin{bmatrix} 0 \\ u_n \end{bmatrix}.$$

$$Y_n = \begin{bmatrix} y_n \end{bmatrix},\ M = \begin{bmatrix} 1 & 0 \end{bmatrix},\ V_n = \begin{bmatrix} v_n \end{bmatrix}$$

The Kalman gain K_n is calculated using the equation:

$$K_n = S^*_{n,n-1} M^T \left[R_n + M S^*_{n,n-1} M^T \right]^{-1}$$

where $Q_n = COV[U_n] = \begin{bmatrix} 0 & 0 \\ 0 & \sigma_u{}^2 \end{bmatrix}$ and $R_n = COV[V_n] = \sigma_v{}^2$

The state transition or prediction equation becomes:

$$X^*_{n+1,n} = \Theta X^*_{n,n}$$

The track update or filtering equation becomes:

$$X^*_{n,n} = X^*_{n,n-1} + K_n (Y_n - M X^*_{n,n-1})$$

If the Estimator receives initialization data from the Receiver, the state vector is initialized using the *position* and *speed* values passed by the neighboring OpenBrick device:

$$X_0 = X^*_{0,-1} = \begin{bmatrix} position \\ speed \end{bmatrix}$$

If the Estimator gets *position* value from the ImageSensor and the Kalman Filter has not been initialized, then the state vector is initialized as:

$$X_0 = X^*_{0,-1} = \begin{bmatrix} position \\ C \end{bmatrix}$$

where C is a constant set to 16 inches per second.

The Covariance matrix $S^*_{n,n-1}$ is initialized as:

$$S_0 = S^*_{0,-1} = \begin{bmatrix} {\sigma_u}^2 & 0 \\ 0 & {\sigma_u}^2 \end{bmatrix}$$

Given Θ, M, R_n, Q_n, n = 0,1,….., $X^*_{0,-1}$ and $S^*_{0,-1}$, the Kalman Filter can be used to calculate estimated position of the person along the aisle by repeating the following steps for n = 0, 1, …. These steps are performed each time the Estimator receives the position of the person in field of view of the host OpenBrick device from the ImageSensor component. This value is nothing but the measured data Y_n.

(a) Compute Kalman Gain using the Formula:

$$K_n = S^*_{n,n-1} M^T \left[R_n + M S^*_{n,n-1} M^T \right]^{-1}$$

(b) Measure Y_n and update estimate using update equation:

$$X^*_{n,n} = X^*_{n,n-1} + K_n (Y_n - M X^*_{n,n-1})$$

(c) Compute covariance of smoothed estimate:

$$S^*_{n,n} = \left[I - K_n M \right] S^*_{n,n-1}$$

(d) Predict using state transition equation:

$$X^*_{n+1,n} = \Theta X^*_{n,n}$$

(e) Compute Predictor Covariance

$$S^*_{n+1,n} = \Theta S^*_{n,n} \Theta^T + Q_{n+1}$$

The final estimated value is obtained from the vector $X^*_{n+1,n}$. This value is taken as the final position of the person in the aisle. Its unit is in inches. It is added to the left range limit of the Estimator and passed on to the DataCollector1 component for further

processing. The left range limit for a particular OpenBrick is specified in the configuration file of application. Its value is dependent on the physical location of the OpenBrick in the sensor network. Thus the Estimator Component passes the position of the person relative to the start of the aisle to the DataCollector1 component.

6.3.4 DataCollector1

The DataCollector1 component receives the actual position of the person in the aisle from the Estimator1 component through shared memory and either records the value in a file as data or sends messages to the neighboring OpenBrick to hand over tracking of that person to its Estimator1. The DataCollector has one input port called ‘Est’ through which it receives this data (Chapter 5, Figure 12). The DataCollector component performs the following tasks:

1. If the person is walking from left to right and his/her position is in the range of overlap with its right neighbor, then the value is sent to the right neighboring OpenBrick device through the wireless network to hand off the tracking of that person to the right OpenBrick device.
2. If the person is walking from right to left and his/her position is in the range of overlap with its left neighbor, then the value is sent to the left neighboring OpenBrick device to hand off the tracking of that person to the left OpenBrick device if it is not disabled.
3. If however, the person is in the range of the current Openbrick device, then the value is saved in a file for reporting.

6.3.5 Estimator2

The Estimator2 component is similar to the Estimator1 component described in section in 6.3.3. However, in addition to tracking people in its own range, Estimator2 also predicts the positions of the people going from its range to the range of a neighboring Openbrick device. The Estimator2 component implements the Kalman Filter in exactly the same manner as Estimator1. However, in addition to the Kalman Filter, Estimator2 also implements a predictor function that is activated once the person enters the range of the right neighboring OpenBrick device which is assumed to be disabled when Estimator2 is executing. The predictor function continues to predict the position of the person till the person remains in the range of the disabled device using the equation;

$$X^{*}_{n+1,n} = \Theta X^{*}_{n,n}$$

```
If(person in range of host OpenBrick)
{
      Use Kalman Filter to estimate position;
      Pass Value returned by Kalman Filter to DataCollector2;
}
else if(person entering range of disabled OpenBrick device)
{
      Estimated_Position = Predictor(Estimated_Position);
      If(Estimated_Position<(right_range_limit+(right_range_limit-
left_range_limit)))
            Pass Estimated_Position to DataCollector2;
      Else
            Stop Estimating;
}

Function Predictor(float position)
{
      return (position + Sampling_Rate * Speed);
}
```

The values for `Sampling_Rate` and `Speed` used in the Predictor function are obtained from the variables used by the Kalman filter. Estimator2 sends all its data to the DataCollector2 component through shared memory. This data includes the estimations of

the position calculated by the Kalman Filter as well as the position calculated by the Predictor function.

6.3.6 DataCollector2

The DataCollector2 component is also similar to the DataCollector1 component. The DataCollector2 component receives the actual position or the predicted position of the person in the aisle from the Estimator2 component through shared memory and either records the value in a file as data or sends messages to the neighboring OpenBrick to hand over tracking of that person to its Estimator component. The DataCollector2 has one input port called ‘Est’ through which it receives this data. The DataCollector2 component performs the following activities:

1. If the person is walking from left to right and his/her position is in the range of overlap with its right neighbor, which is disabled, or beyond its own right range limit, the value is recorded by the DataCollector2 component in a file with the suffix “ESTIMATE”.
2. If the person is walking from right to left and his/her position is in the range of overlap with its left neighbor, then the value is sent to the left neighboring OpenBrick device to hand off the tracking of that person to the left OpenBrick device.
3. If however, the person is in the range of the current Openbrick device, then the value is saved in the file for reporting.

During experiments, the configuration of the Aislemonitor application executing on nodes adjacent to the disabled OpenBrick device is dynamically changed from the first to the second. Initially the ImageSensor, Receiver, Estimator1 and DataCollector1 components are made to execute on all the OpenBrick devices. One of the OpenBrick devices is then shut down. This triggers reconfiguration of the OpenBrick device located to the left of the device that has been disabled. Reconfiguration involves just switching the Estimator1 and DataCollector1 components with the Estimator2 and DataCollector2 components and the ‘re-wiring’ of all the shared memory between all the components. The ImageSensor and Receiver components are not affected by the reconfiguration. They only reconnect their ports to the newly started components.

CHAPTER VII

SOFTWARE INFRASTRUCTURE FOR RECONFIGURATION

The software reconfiguration process occurs at both the base station and the sensor nodes. During the design phase, the applications are modeled in the SNRAMoLa modeling paradigm on the base station. Once the application is deployed the tasks of design space exploration, communication of the configuration to the sensors, monitoring the sensors and updating QoS parameters in the models are performed in a cyclical manner. The software infrastructure required to perform these tasks consists of components that execute on the base station as well as on individual OpenBrick devices. All applications using this infrastructure are based on the Asynchronous Data Flow model of computation.

As shown in Figure 10 (Chapter 4) the reconfiguration architecture consists of the modeling environment SNRAMoLa, which enables the user to model components along with their alternatives and associated constraints in GME. During the reconfiguration process, the application model is converted to a format acceptable to the design space exploration tool DESERT by the SNRAMoLa to DESERT Interpreter. The converted data is fed to DESERT as an XML file. DESERT reads from the file, applies the constraints present in the model and generates another XML file. The DESERT to Configurator interpreter then uses the output generated by DESERT and the SNRAMoLa model of the application to generate a configuration file for each Sensor object present in the SNRAMoLa model. Each Sensor object in SNRAMoLa represents an OpenBrick

device in our example. The configuration files list out all the components that are to be activated on the sensor and also the connections between the ports of the components.

Each configuration file is broken down into UDP packets and then sent to the OpenBrick device connected to the base station by the controller application running on the base station. A forwarding program called Packet Forwarder on that OpenBrick device forwards the packets to the corresponding OpenBrick devices over the ad hoc wireless network.

The Configurator program executing on each OpenBrick device receives the configuration file and carries out the actual reconfiguration by stopping, starting and re-wiring application components on the OpenBrick device.

During initialization, each process establishes a connection with the Configurator using shared memory. This connection is used by the Configurator to send reconfiguration commands to the processes. The connection with the Configurator is established using the Communicator component, which is part of the software infrastructure and invoked by each process during initialization.

Monitor components perform the task of monitoring the health of the sensor network. The Monitor executes on each sensor and communicates with the Global Constraint Monitor executing on the base station. The Global Constraint Monitor updates the attribute values for critical QoS parameters such as power in the SNRAMoLa Component models. A change in the value of these attributes drives the Controller program which starts the reconfiguration process on the base station.

The next sections of the chapter explain the different components of the software reconfiguration architecture. The first section explains the Controller program followed

by the SNRAMoLa to DESERT interpreter. Section 7.3 explains the DESERT to Configurator interpreter. Section 7.4 explains the layout of the Configuration files passed by the Controller to the Configurator, Section 7.5 explains the Packet Forwarder and Section 7.6 explains the Configurator. Section 7.7 explains the Communicator component while Sections 7.8 and 7.9 explain the Monitor and the GCM components respectively.

7.1 Controller Program

The controller program performs the reconfiguration process on the base station. It is invoked by the Global Constraint Monitor after it updates the values of the QoS attributes in the SNRAMoLa application model.

The controller program first invokes the SNRAMoLa to DESERT interpreter which converts the SNRAMoLa models into a format compatible with DESERT input and saves the model as an XML file. The controller then invokes DESERT which takes the XML file saved by the controller and evaluates the constraints on the model using Ordered Binary Decision Diagrams [39]. Upon resolving the constraints, DESERT generates another XML file which contains a binding of each OR-Decomposed node in the AND/OR decision tree with one of its children. The controller then invokes the DESERT to Configurator interpreter which takes the DESERT output along with the SNRAMoLa model to produce a configuration file for each Sensor object declared in the SNRAMoLa model.

The controller then sends all the configuration files as UDP packets to the Packet Forwarder program, which executes on the OpenBrick device connected to the base station by a wired connection. The Packet Forwarder relays the UDP packets to the

respective OpenBricks over the ad hoc wireless network. After sending all the files, the controller sends the "RECONFIGURE" control signal to the Packet Forwarder which again relays it to the respective OpenBricks. The controller addresses the packets to the Packet Forwarder using the IP address and Port number specified in the SenNet.conf configuration file.

Each data packet sent to the Packet Forwarder contains a header, which includes the destination IP Address and listening Port number. The listening Port number is obtained from the SenNet.conf configuration file during initialization while the IP address of the destination OpenBrick device is specified in the SNRAMoLa model as the name of the Sensor object. Example control signals sent to the Packet Forwarder are shown in the following window.

```
"192.168.10.2:4952:CONFNO 2
COMPONENT /usr/local/component1/component1
COMPONENT /usr/local/component2/component2
LINK /usr/local/component1/component1 OutPort
/usr/local/component2/component2 InPort"

"192.168.10.2:4952:RECONFIGURE"

"192.168.10.3:4952:STOP"
```

7.2 SNRAMoLa to DESERT Interpreter

The SNRAMoLa to DESERT Interpreter converts the SNRAMoLa application model to a format compatible with DESERT by building an AND-OR tree whose node elements map directly to the objects modeled in SNRAMoLa.

The design space for an application modeled in SNRAMoLa can be formally defined as follows. A SNRAMoLa model Ra is a tuple (T_g, C), where C is the set of SNRAMoLa Conditions, and T_g is a tree (N, E). The vertex set N of T_g is the set of

SNRAMoLa objects declared in the modeled application (i.e. SensorFolder, Sensors, Components and Choices). The directed edge set E of T_g represents containment relation between the modeling elements.

$$\forall v_1, v_2 \in N \quad v_2 \in children(v_1) \; iff \; \exists e = (v_1, v_2) \in E \tag{1}$$

where $children(n)$ is the set of objects contained in an object n. SNRAMoLa objects are defined as $Sf \subset N$, $Cmp \subset N$, $Ch \subset N$ and $Sen \subset N$ as disjoint set of objects of type *SensorFolder, Components, Choices,* and *Sensors* respectively. The SensorFolder set *Sf* is a singular set and is always mapped as the root of the AND-OR tree. A Choice exhibits OR-decomposition semantics, while a SensorFolder and Sensors exhibit AND-decomposition semantics. SNRAMoLa Components are characterized with properties, over which the constraints expressed in the Condition objects are evaluated. An example of such a property is the power available to a particular Component.

A DESERT Constraint $c \in C$ is a tuple $(cons, ctx)$, where *cons* is the constraint expression, written in a variant of OCL, and $ctx \in N$, is the context of the constraint (referred to using the OCL keyword 'self' in the constraint expression) The SNRAMoLa model is mapped onto a single DESERT *Space*. The bijection $Ra2Des : N \leftrightarrow S$ maps from objects in *Ra* to elements in DESERT, where S is the set of DESERT elements. The following holds under this mapping:

$$\forall v_1, v_2 \in N \quad v_1 \in children(v_2) \Leftrightarrow Ra2Des(v_1) \in \chi(Ra2Des(v_2)) \tag{2}$$

The decomposition attribute of a DESERT element is defined as follow:

$$\forall de \in S \tag{3}$$

$$decomposition(de) = \begin{cases} true & Ra2Des^{-1}(de) \in Sf \\ true & Ra2Des^{-1}(de) \in Sen \\ false & Ra2Des^{-1}(de) \in Ch \end{cases}$$

The mapping of SNRAMoLa Conditions to DESERT involves the mapping of the SNRAMoLa Condition context onto its projection element under the $Ra2Des$ bijection. The properties of SNRAMoLa Components are mapped to properties in the corresponding DESERT element, with the values appropriately associated. Once the mapping from SNRAMoLa onto DESERT is complete for an application, DESERT prunes and explores the design space.

The nodes of the Desert Space have a direct mapping to the Sensor, Choice and Component objects from the application model. The root of the Space always maps to the SensorFolder object in the SNRAMoLa model. The children of the root node in the Space represent the Sensor objects from the SNRAMoLa model. The children of each of these nodes represent non-reconfigurable Component objects and Choice objects. The nodes representing the non-reconfigurable Components are leaf nodes and do not have any children. The children of the nodes mapping to the Choice objects represent reconfigurable Component objects declared in the Choice objects in the SNRAMoLa application models. The Condition objects are included in the DESERT Space as Constraints with their Context associated with the OR-decomposed nodes, which map to the Choice objects. The SNRAMoLa to DESERT interpreter performs this mapping when

invoked by the controller program and generates an XML file, which is fed to DESERT as input.

7.3 DESERT to Configurator Interpreter

A DESERT output configuration contains a binding for each OR-decomposed element in the DESERT Space to a direct child of that element. The binding represents the resolution of the design choice represented by the OR-decomposition. The resulting configuration file has all design decisions resolved. All the Choice objects in SNRAMoLa which map onto the OR-decomposed elements in DESERT are resolved by the DESERT pruning process and bound to reconfigurable Component objects declared inside them, which map to the child elements of the OR-decomposed elements in DESERT.

The DESERT to Configurator Interpreter reads from the SNRAMoLa model file (an .mga file) and the DESERT output file (an .xml file) and creates individual configuration files for each Sensor object declared in the SNRAMoLa model. These configuration files are identified by the IP addresses assigned to the OpenBrick devices which are also the names identifying the Sensor objects in the SNRAMoLa model.

For each Sensor object in the SNRAMoLa model, the interpreter first updates the configuration number in the corresponding configuration file. If a file does not exist, then a file is created with the configuration number value equated to 1. The interpreter then writes to the file the paths of all the Components declared or referred to in the Sensor objects. For each Choice object in the SNRAMoLa model, the interpreter refers to the corresponding OR-decomposed element in the output file generated by DESERT and

writes the path of the Component object, referred to by the child of the OR-decomposed element bound to it, to the file.

After traversing through all the Component objects, the interpreter goes through all the DataFlow connections declared in the SNRAMoLa model. Each DataFlow connection connects the OutPort object of a Component with the InPort object of another and signifies the asynchronous flow of data between the components. These connections are written to the configuration file as Links along with their source and destination Component object name and port information. These configuration files are then physically transported to the respective OpenBrick devices over the wireless network by the controller program.

7.4 Configuration Files

The DESERT to Configurator Interpreter generates a configuration file for each Sensor object declared in the SNRAMoLa model of the application. An example Configuration file is shown in the following window.

```
CONFNO 1
COMPONENT /root/sachin/aislemonitor/receiver/receiver
COMPONENT /usr/local/motion/estimator
COMPONENT /root/sachin/aislemonitor/application/datacollector
LINK /root/sachin/aislemonitor/receiver/receiver OutPort
/usr/local/motion/motion InPort
LINK /usr/local/motion/motion OutPort
/root/sachin/aislemonitor/application/datacollector InPort
END
```

The first line of the file always contains the configuration number which indicates the number of times the OpenBrick device has been reconfigured. It is incremented each time the file is updated. The file then contains the individual Components that are included in the application graph after the resolution of the constraints by DESERT using

the keyword ‘COMPONENT’. The Components are identified by the path of the executable that is invoked to start them on the OpenBrick device. This information is included in the Path attribute of the Component object in the SNRAMoLa model. The lines beginning with the keyword ‘LINK’ represent the DataFlow connections from the SNRAMoLa models. The LINKs are identified by the string composed of the Path in the source Component object, the name of the OutPort object on the source Component object, the Path in the destination Component object and the name of the InPort object on the destination Component object.

7.5 Packet Forwarder

The Packet Forwarder component executes on the OpenBrick device connected to the base station by a wired 802.3 Ethernet connection. The Packet Forwarder is just a relaying program that receives messages coming from the base station and sends them over to the destination OpenBrick devices over the ad-hoc wireless network. The incoming messages include the destination IP address and the listening port number of the destination OpenBrick devices. The Packet Forwarder obtains the destination IP address and port from the incoming message and sends only the message text minus the addressing information to the destination OpenBrick devices. The Packet Forwarder also acts as a relaying program when messages are sent to the base station by the OpenBrick devices. It performs the same activity, except sends the message in the opposite direction. In our testbed, the base station was connected to one OpenBrick device through a wired LAN connection. However, this is not a requirement of the software infrastructure. If a wireless enabled base station is placed in the wireless range of one of the sensor nodes, it

can directly communicate with each Sensor node over the ad hoc sensor network and will not need the Packet Forwarder.

7.6 Configurator

The Configurator is the most important component of the software reconfiguration infrastructure. A copy of the Configurator executes on all the sensor devices. The Configurator implements the reconfiguration infrastructure by maintaining two link-list data structures and a memory ID counter.

7.6.1 Process Data Structure

The Processes link-list stores information about all the processes that are currently executing on the OpenBrick device and is composed of Process structures. When a new component is added in the configuration file, the Configurator adds a new Process structure to the Processes link-list before executing it. One Process structure element in the Processes link-list represents one process running on the OpenBrick. The Process structure is shown in the following window.

```
Struct Process
{
      int configuration_no;
      p_id process_id;
      char process_name[200];
      int shared_memory_id;
      char *shared_memory_pointer;

      struct Process *next;
};
```

Each element in the Processes link-list stores the configuration number of the process, the process ID of the process, the name of the process which is read from the configuration file, the memory ID of the shared memory used by the Configurator to

communicate with the process and the pointer to this shared memory. The configuration number just identifies the current configuration of the process.

The Configurator program uses shared memory to communicate with each process. The information stored in the Processes link-list is used by the Configurator to communicate reconfiguration commands to individual processes. When a Configurator needs to send a message to a process, (1) it gets the record for that process from the Processes link-list using the process name, (2) writes the message to the shared memory using the memory pointer and (3) then signals the process to read the shared memory by using the process ID in the UNIX kill command. The Configurator interrupts application processes using the kill command by signaling them to interpret the reconfiguration commands.

7.6.2 Link Data Structure

The Links link-list stores all the information about the shared memory that is used to pass data between two processes. The Links link-list is composed of Link structure elements. Each element in the Links link-list is identified by the memory name, which is just a concatenation of the names of the two processes that it connects. In addition to the name of the link, each element of the link-list stores the configuration number, memory ID, a pointer to the shared memory, a pointer to the source process, a pointer to the destination process, the name of the OutPort object and the name of the InPort object. The Link structure is displayed on the following page.

```
Struct Link
{
      char link_name[200];
      int configuration_no;
      int shared_memory_id;
      char * shared_memory_pointer;

      struct Process *source_process;
      char outport_name[100];

      struct Process *destination_process;
      char inport_name[100];

      struct Link *next;
}
```

The Configurator maintains a memory ID counter that is incremented every time it allocates a new shared memory. This memory ID is passed as an argument to the processes during initialization and reconfiguration using which the processes ‘rewire’ to connect to the appropriate shared memory. The pointer to the shared memory points to the allocated memory chunk that is used by both the source and destination processes to exchange information. The pointer to the source process points to the Process structure element in the Processes link-list which is the source of the Data Flow connection and the pointer to the destination process points to the element in the Processes link-list which is the destination of the Data Flow connection. Each Link structure also stores the names of the OutPort and the InPort objects of the source and the destination process respectively. The Configurator does not write to the shared memory pointed to by the Link structures but creates and maintains it so that the source and destination processes can just connect to it to exchange information.

7.6.3 Flow of Control

During initialization, the Configurator opens a socket to listen for incoming signals from the base station. After initialization, the Configurator goes in an infinite loop where it continues to listen for new messages on the open port coming from the base station. Upon receiving a message, it performs reconfiguration activities and then goes back to listening for new messages. The Configurator once started, can be stopped only by sending a STOPLISTENING signal to its listening port. The messages received by the Configurator can be classified into four types – new configuration file, FILERECEIVED command, STOPLISTENING command and RECONFIGURE command. The Configurator follows the following algorithm:

1. Listen for incoming message;
2. IF data is received, copy it to the variable called message;

```
IF ((STOPLISTENING || FILERECEIVED || RECONFIGURE) NOT IN
message)
GO TO step 3;
ELSE GO TO step 4;
```

3. If Configurator receives a message which does not contain any of the three commands, it assumes the message to be a new configuration file. If the configuration file is too big, it is broken down and sent using multiple packets by the Controller program at the base station.

```
IF(message == first packet received)
  Create a new configurator.conf file and write the message to
the file;
ELSE
  Open the existing configurator.conf file and append the
message to the file;

CLOSE(configurator.conf);
GOTO step 1;
```

4. The command string FILERECEIVED indicates the end of the configuration file. If the Configurator receives another message that does not match with any of the commands, then it considers this as the beginning of a new configuration file.

```
IF (message==FILERECEIVED)      {
  Set flags such that the next message which does not contain
the command words {FILERECEIVED, STOPLISTENING, RECONFIGURE)
is taken as the first packet of a new file;
GOTO step 1;
}
ELSE
  GOTO step 5;
```

5. Upon receipt of the STOPLISTENING command from the base station, the Configurator sends a STOP control signal to all executing processes of the application using the information stored in the Processes link-list, closes its own listen port, frees all allocated memory for the link-lists and the shared memories, exits out of the infinite loop and terminates.

```
IF (message == STOPLISTENING)   {
  FOR EACH Process IN Processes Link-List   {
        WRITE ("Stop", Process → shared_memory);
        Kill (Process → process_ID, SIGUSR1);
  }
  FREE (Processes link-list);
  FREE (Links link-list);
  GOTO step ;
}
ELSE
  GOTO step 6;
```

6. The RECONFIGURE command triggers the reconfiguration process of the Configurator. Upon receipt of this command, the Configurator reads from the configuration file, configurator.conf, which is also received from the base station. If the configuration number of the file is different than the one from the previous file, the Configurator reads the file sequentially; else it goes back

in the listening mode. The configuration number is identified by the CONFNO attribute in the configuration file.

```
IF (new_configuration_filenumber ==
old_configuration_filenumber) {
  CLOSE configuration.conf file;
  GOTO step 1;
}
ELSE
  GOTO step 7;
```

7. Read from the configuration.conf file and copy the data to the File_Word variable.

```
int link_flag=0;
IF (File_Word==COMPONENT) GOTO step 8;
ELSE IF (File_Word==LINK) {
  IF(link_flag==0) {
    Link_flag=1;
    GOTO step 9;
  }
  ELSE
    GOTO step 10;
}
ELSE IF (File_Word==END) {
  IF(link_flag==0) GOTO step 9;
  ELSE
  {
    CLOSE (configurator.conf);
    GOTO step 1;
  }
}
```

8. For each COMPONENT keyword present in the file, the Configurator reads the component path identifying the process into the variable ‘process_path’. The Configurator then checks if the corresponding process is already executing on the OpenBrick by looking it up in the Processes link-list. If the process is found, the Configurator just updates its configuration number by the CONFNO value. If the process is not found then the Configurator adds the process name/path to a temporary string array ‘process_names’.

```
Struct Process new_process IS NULL;
IF (new_process = PROCESS_EXISTS(process_path)    {
  new_process → configuration_no = new_configuration_no;
ELSE {
  COPY(process_path, process_names);
GOTO Step 7;
```

9. On reading the first LINK keyword or the END keyword from the file, the Configurator traverses through the Processes link-list and for each process element whose configuration number is not equal to CONFNO, it sends the control command STOP to the corresponding process using the shared memory and the process ID and then deletes that element from the Processes link-list. The receipt of the STOP command from the Configurator causes the process to terminate. After switching off all the processes that are not included in the new configuration file in this manner, the Configurator goes through the temporary string array, which stores the names/paths of the new processes. For each element in this array, the Configurator adds a new Process element to the Processes link-list and executes the new process using the UNIX exec command and the path obtained from the configuration file. The Configurator passes the memory ID and the process name or path as command line parameters to the new process, which it uses to connect to the shared memory used by the Configurator to send control signals.

```
FOR EACH Process IN Processes
{
  IF(Process→configuration_no < new_configuration_no)
  {
        WRITE("STOP",Process→shared_memory);
        Kill(Process→process_ID,SIGUSR1);
  }
}
FOR EACH process_name IN process_names
{
  Struct Process *new_process=NULL;
  Allocate memory for new_process;
  new_process.process_name=process_name;
  //process_name obtained from configuration file
  Allocate Shared memory;
  ADD_ELEMENT(Processes, new_process);
  execlp(new_process.process_name, new_process.process_name,
new_process.shared_memory_ID)
}
IF(link_flag==1)
  GOTO step 10;
ELSE    {
  CLOSE(configurator.conf);
  GOTO step 1;
}
```

10. After switching on new processes, the Configurator traverses through the list of links in the configuration file. For each LINK in the configuration file, the Configurator reads its corresponding name in the 'link_name' variable and then checks if it already exists in the Links link-list. If the record is found, the Configurator just updates its configuration number. If no such link exists, the Configurator adds a new Link element to the Links link-list, allocates memory, points the shared memory pointer in the new Link structure to the allocated memory, and initiates all the other fields in the Link structure. Each Link element in the Links link-list is identified by a name, which is formed by concatenating the names of the source and the destination processes. After adding a new element to the link-list, the Configurator communicates with both the source process and the destination process using the shared memory

accessible through their respective records in the Processes link-list. The Configurator sends message 1 to the source process over the shared memory and sends message 2 to the destination process over the shared memory. The messages are shown below.

```
1]
LINK:memoryID:OutPortName:SourceProcessID:DestinationProcessID

2]
LINK:memoryID:InPortName:SourceProcessID:DestinationProcessID
```

```
Struct Link *new_link IS NULL;
IF (new_link = LINK_EXISTS(link_name) {
  new_link → configuration_no = new_configuration_no;

ELSE {
  Allocate memory for new_link;
  new_link.link_name=link_name;
  //link_name obtained from configuration file
  Allocate Shared memory;
  ADD_ELEMENT(Links, new_link);
  WRITE(new_link→source_process→shared_memory,"
LINK:memoryID:OutPortName:SourceProcessID:DestinationProcessI
D");
  Kill(new_link→source_process→process_id,SIGUSR1);
  WRITE(new_link→destination_process→shared_memory,"
LINK:memoryID:InPortName:SourceProcessID:DestinationProcessID
");
  Kill(new_link→destination_process→process_id,SIGUSR1);
}
GOTO Step 7;
```

The processes use the Communicator component to interpret the control commands received from the Configurator.

7.7 Communicator

The Communicator component consists of a library of functions that enable individual processes to communicate amongst themselves and with the Configurator. The Communicator is included in every process executing on the OpenBrick.

The Configurator communicates with a process by writing the control message on the shared memory and then by signaling the process using its process ID and the UNIX kill command. The Configurator uses the SIGUSR1 signal to interrupt the receiving process. Communication between two processes also occurs along similar lines. The sending process writes the data to the shared memory and then signals the receiving process using its process ID and the UNIX kill command. However, in inter-process communication, the sending process uses the SIGUSR2 signal to interrupt the receiving process. Communication between processes is modeled using Data Flow connections and input and output ports represented by InPort and OutPort objects in the SNRAMoLa model. The communication between the process and the Configurator is not modeled in SNRAMoLa and is part of the software infrastructure.

7.7.1 Connection Data Structure

The Communicator implements the communication between the process and the Configurator using shared memory. The Communicator component maintains a Connection data structure which stores the name of the link connecting the process to the Configurator, memory ID and a pointer to the shared memory which is used to actually pass the data. This link is not modeled in SNRAMoLa as it is not a part of the application.

```
Struct Connection
{
      char connection_name[200];
      int configuration_no;
      int shared_memory_id;
      char * shared_memory_pointer;

      p_id source_process_id;
      p_id destination_process_id;

      struct Connection *next;
{
```

7.7.2 Communicator Initialization

During initialization, the main function of the process invokes the Communicator_init method implemented in the Communicator component. This initializes the Connection data structure. The name of the Connection data structure is set to the name of the process and the memory ID is set to the memory ID, both passed as a command line argument to the process by the Configurator. The Communicator uses this memory ID to point the pointer to the correct shared memory which is already allocated and maintained by the Configurator in its Processes link-list.

In addition to maintaining the Connection data structure, the Configurator also implements a signal handler function to handle the SIGUSR1 signal. When the process is interrupted by the Configurator using the SIGUSR1 signal, the signal handler function is invoked and it interprets the data sent by the Configurator after reading it from the shared memory. Upon receiving the STOP message from the Configurator, the signal handler terminates the executing process by issuing the kill SIGKILL command.

7.7.3 Connections link-list

In addition to the single Connection data structure, the Communicator also maintains a link-list of the Connection data structure called the Connections link-list. This list maintains information about all the ports of the process. These ports directly map to the InPort and OutPort objects modeled in the SNRAMoLa model of the corresponding Component object and are used by the processes to exchange information. Each Connection element in the link-list stores the name of the port, memory ID, a pointer to the shared memory, the process ID of the source process and the process ID of the destination process. The name of the port is used as the unique identifier of the element in the link-list. The port name is the same as the name of the corresponding InPort or OutPort object in the SNRAMoLa model. The communication between two processes is modeled using OutPort, InPort and Data Flow objects in the SNRAMoLa model. It is implemented using shared memory and the Connection structures in the actual processes.

7.7.4 Communicator Algorithm

The Communicator is not a separate process like the Configurator. It initializes when the process is invoked by the configurator and then stays dormant. The Communicator performs the following tasks when interrupted by the Configurator using the SIGUSR1 signal.

Upon receiving the –

‘LINK:*memoryID*:*PortName*:*SourceProcessID*:*DestinationProcessID*’

message followed by the SIGUSR1 interrupt signal from the Configurator, the signal handler function searches the Connections link-list for an element with its port name

matching the *PortName* passed by the Configurator. If such an element is found, it indicates a port by the same name already exists but is currently linked with some other shared memory. The Communicator then disconnects the port from the shared memory and connects it to the shared memory identified by the *memoryID* sent by the Configurator. The Communicator also updates the source process ID and the destination process ID fields of the element. If an element with same port name is not found in the Connections link-list, the Communicator adds a new element to the link-list and updates all its fields with the values sent by the Communicator.

Upon receiving the – `'STOP'` message from the Configurator, the Communicator terminates the current process by issuing the `kill (my_process_id, SIGKILL)` UNIX command.

When a process wants to send a message to another process, it gets the required Connection element from its Connections link-list using the port name. The port name used by the Communicator is the same as the name of the corresponding OutPort object in the SNRAMoLa model. The process utilizes the pointer stored in the Connection element to write the data to the shared memory. The process then utilizes the destination process ID stored in the Connection element along with the UNIX kill command and SIGUSR2 signal to interrupt the destination process.

If a process has any InPorts objects, it should always implement a signal handler function for the SIGUSR2 signal. This function is invoked upon receiving a SIGUSR2 interrupt from another process. The function should go through all the Connection elements in the Connections link-list and for each matching InPort object of the process, read the data from the shared memory.

The Communicator thus implements all the Data Flow connections modeled in the SNRAMoLa model and can change the connections dynamically upon receiving such a command from the Configurator.

7.8 Monitor

The Monitor components execute on each OpenBrick device and monitor the health of the sensor network. For the purpose of this experiment, the Monitor is integrated with the application. Each Monitor monitors the health of its right neighboring OpenBrick device. The left and right neighbors are specified in the aislemonitor.conf file. The Monitor is implemented using a simple algorithm in which each OpenBrick device sends a beacon message to its left neighbor every few seconds (10 seconds). If an OpenBrick device does not receive a message from its right neighbor for a fixed amount of time, it sends a message to the Packet Forwarder which relays it to the GCM on the base station and informs it that the right neighbor has failed. The packet sent to the Packet Forwarder contains the following message:

'*BaseIP*:*BasePort*:DOWN:*RightNodeIP*:*SelfIP*'

The Packet Forwarder strips the message of the *BaseIP* and the *BasePort* and uses that information to relay the remainder of the message to the GCM.

7.9 Global Constraint Monitor (GCM)

The GCM updates the QoS parameters like power in the SNRAMoLa models of the application and then invokes the Controller program, which performs the reconfiguration process on the base station. The GCM executes on the base station and receives messages from the Packet Forwarder. If a particular OpenBrick device fails to

send the beacon messages to its left neighbor, the left neighbor sends a message to the GCM. The format of the message is described in the previous paragraph. The GCM then toggles the values of the power attributes of the Components declared in the Choice objects in the SNRAMoLa model of the application for the Sensor object identified by the *SelfIP* part of the message or the sender OpenBrick device. After updating the power attribute, the GCM invokes the Controller program which carries out the entire reconfiguration process on the base station and then sends the new configuration files to the OpenBrick devices where the actual reconfiguration takes place with the switching of the executing processes.

CHAPTER VIII

EXPERIMENTS AND RESULTS

A series of experiments were performed to evaluate the Aislemonitor application (described in Chapter 6) and the software reconfiguration infrastructure (described in Chapter 7). These tests were designed to highlight the need for software reconfiguration and its application using our infrastructure.

This chapter describes all the experiments that were carried out. The first section describes the test cases that were used in the evaluation. The next section analyzes the results of the tests and the final section provides a summary of the results.

8.1 Test Case Design

Multiple tests involving people walking in the aisle in groups of one, two and three were performed. The tests were designed to record the position and speed of the people walking in the aisle along with the time. Data was recorded on all the OpenBrick devices and was later integrated and plotted to generate graphs indicating the movement of the people in the aisle.

The tests that involved more than one person walking in the aisle were designed such that at any given time, more than one person was in the range of the sensor network for some interval of time but never in the range of the same OpenBrick device. Care was taken to keep only person in the range of an OpenBrick device at any given time.

In order to measure the time required for reconfiguration of components, the timestamps at various instances in the reconfiguration process were recorded.

8.2 Evaluation of Configuration 1 of Aislemonitor Application

The Tests 1 – 3 were used to evaluate the first configuration of the Aislemonitor application. The first configuration of Aislemonitor (described in Chapter 6) was deployed on all the OpenBrick devices. The first configuration includes components that can only detect and track a person in the range of the host OpenBrick device.

8.2.1 Configuration 1 Tests

1. Test 1 involved two people with one person walking from left to right along the entire length of the aisle followed by two people walking back.
2. Test 2 involved two people with two people walking from left to right followed by one person walking from right to left.
3. Test 3 involved three people walking from left to right along the entire length of the aisle and only two people coming back.

8.2.2 Configuration 1 Results

The results of the tests for evaluating the first configuration are plotted in the graph shown in Figure 20. The x axis in the graph represents the time at which the readings were taken. The y axis represents the position of people in inches from the start of the aisle, where the first OpenBrick device is located. Figure 20 shows the positions for the two people tracked by the application. P1-LR and P2-LR indicate persons 1 and 2 walking from left to right in the aisle one after the other. The second person enters the range of the sensor network before the first person moves out of it. P1-RL indicates the first person walking from the right to the left in the aisle. The graph indicates an almost

linear curve for each person walking in the aisle. This indicates a near constant speed used by the people while walking.

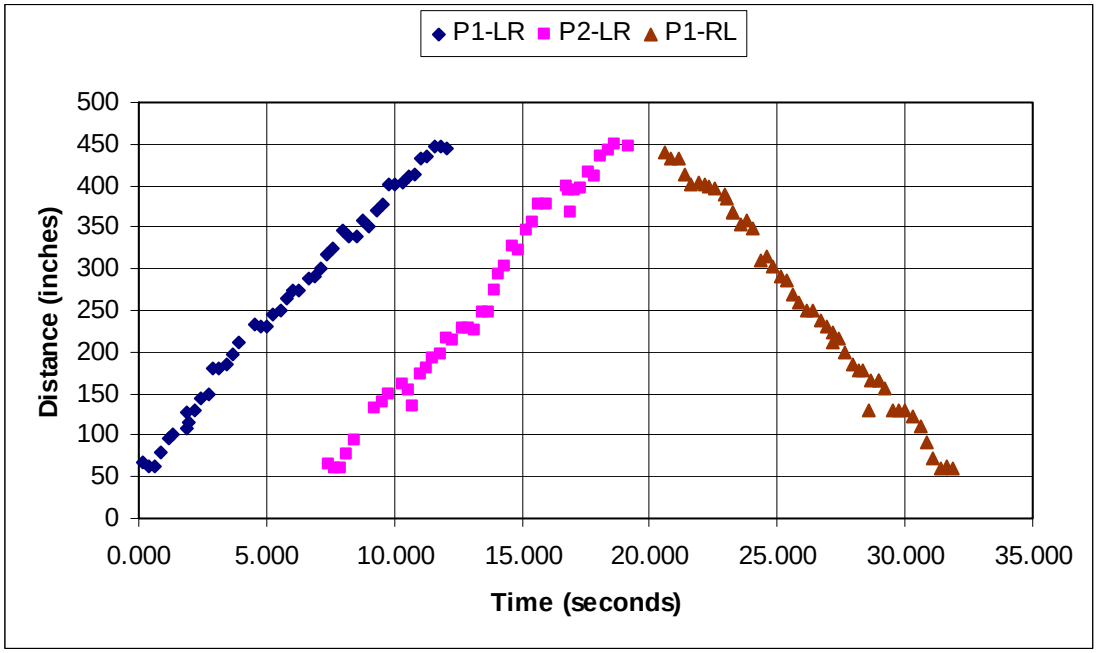

Figure 20 – Aislemonitor Configuration 1 results

8.3 Evaluation of Configuration 2 of Aislemonitor Application

Tests 4 and 5 were designed to evaluate the second configuration of the Aislemonitor application. After the third test, the fourth OpenBrick device was switched off. This triggered software reconfiguration on the third OpenBrick device. The second configuration includes components that in addition to tracking people in their own range, can also track people in the range of the right neighboring OpenBrick device. These tests were aimed at generating some 'prediction' data in the range of the disabled OpenBrick device.

8.3.1 Configuration 2 Tests

1. Test 4 involved one person walking from left to right and then coming back.
2. Test 5 involved two people walking from left to right and then coming back.

8.3.2 Configuration 2 Results

The results of the tests evaluating the second configuration are plotted in the graphs shown in Figures 21 and 22. During the tests, the fourth OpenBrick device was turned off. The second configuration of the Aislemonitor application was then dynamically deployed on the third device to predict the position of the person in the range of the fourth device. Figure 21 displays the graph generated by plotting the position readings from the third and the fifth OpenBrick devices with no prediction capability in the third device. The gap created in the sensing range as result of this is clearly seen.

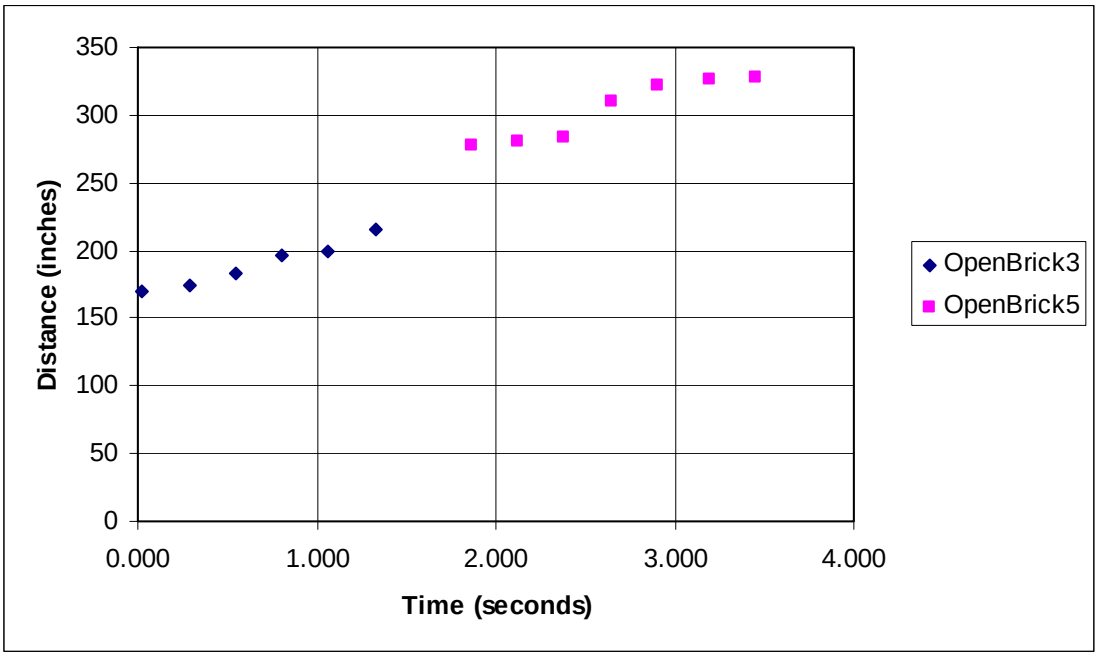

Figure 21 – "Gap" in Sensing Range

Figure 22 displays the same graph, but this graph also includes the prediction readings from the third OpenBrick device. The figure shows the gap filled in by the predicted values of the person in the gap by the third OpenBrick device.

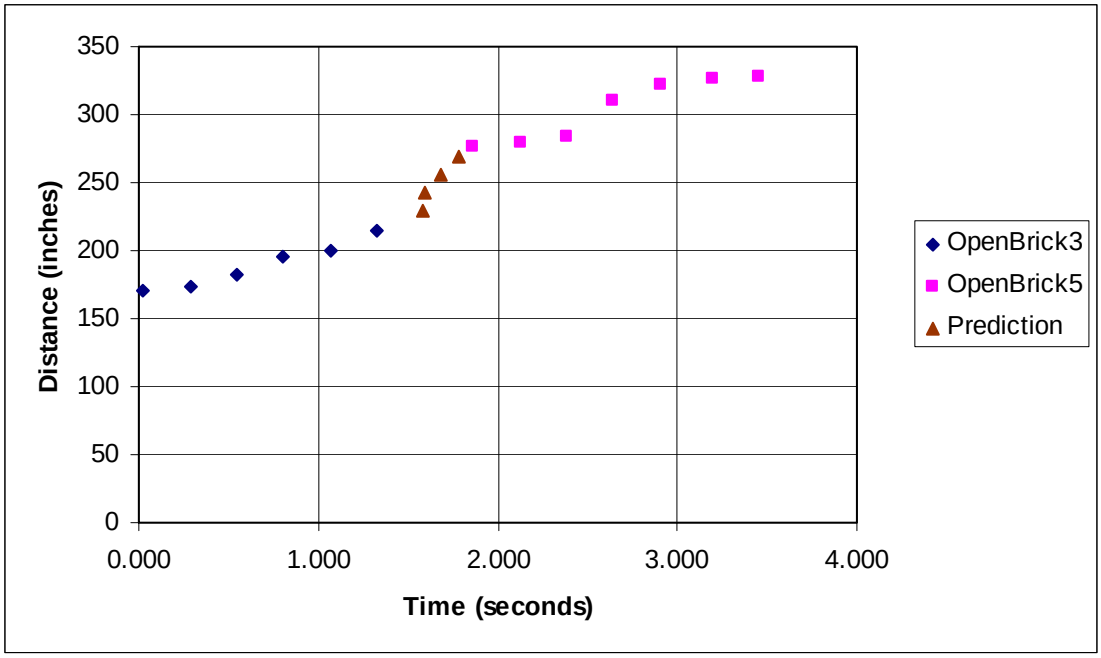

Figure 22 – "Gap" in Sensing Range Filled by Prediction

8.4 Tests Demonstrating need for Software Reconfiguration

Tests 6 and 7 were performed to demonstrate the need for performing software reconfiguration. In these tests, the fourth OpenBrick devices was shut down and software reconfiguration was not performed on the third OpenBrick. This created a sensing 'gap' in the network. All the devices executed the first configuration of the Aislemonitor application.

8.4.1 Tests

1. Test 6 involved one person walking from left to right and then coming back.
2. Test 7 involved two people walking from left to right along the aisle and then coming back.

8.4.2 Results

Tests 6 and 7 were performed to demonstrate the usefulness of the reconfiguration architecture. The gap created in the sensing range of the sensor network due to the failure of one of the OpenBrick device can be reduced significantly by dynamically reconfiguring the components executing on the neighboring OpenBrick devices. Figure 23 displays the comparisons of all the tests.

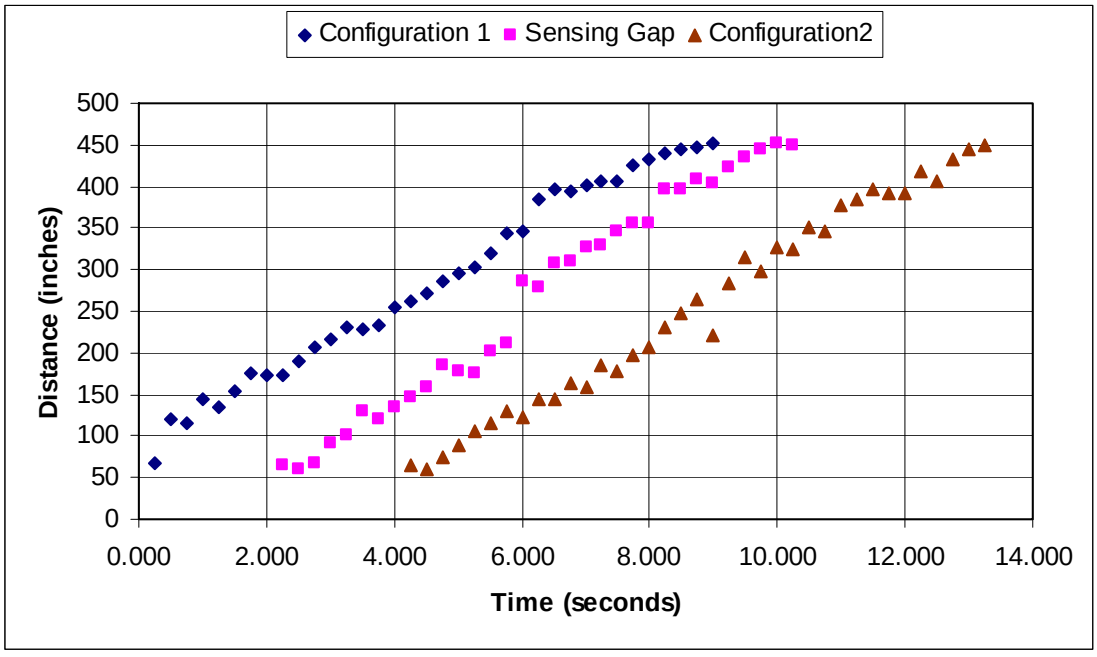

Figure 23 – Comparison of Tracking Algorithms

The line representing ‘Configuration 1’ indicates a person walking from left to right with all the devices working. The line representing ‘Sensing Gap’ highlights the sensing gap generated when there is no reconfiguration. The line representing ‘Configuration 2’ indicates the person walking when the fourth OpenBrick device has been disabled and the third OpenBrick device is executing the second configuration.

8.5 Evaluation of the Software Reconfiguration Architecture

The dynamic switching of components executing on the third OpenBrick device after the disabling of the fourth OpenBrick was evaluated by recording the times at which various reconfiguration activities took place. The time required for the whole reconfiguration process was of particular interest. During the tests, time was recorded when (1) the Monitor on the third node sent a message to the GCM, which triggered the reconfiguration process, (2) new configuration files were sent by the base station to the third OpenBrick device, (3) configuration files were received by the Configurator on the third device, (4) reconfiguration commenced on the third device, and (5) reconfiguration was completed.

The Configurator on the third OpenBrick device carried out dynamic software reconfiguration. After the fourth OpenBrick device was disabled, the Monitor on the third device waited for an interval of 10 seconds before sending a message to the GCM on the base station. This waiting period was set during the initialization of the application. When a message was received by the GCM, it updated the SNRAMoLa models of the application and invoked the controller program. The Controller program performed software reconfiguration on the base station and exited after sending the new

configuration files to the OpenBrick devices. The reconfiguration process from the receipt of message from the OpenBrick to the dispatch of a new configuration file to it took 10 seconds. The Configurator upon receipt of a new configuration file performed the actual software reconfiguration on the third OpenBrick device. This process took another 8 seconds. The total reconfiguration process took 28 seconds, if the time required for fault detection is also considered. If it is not considered, then the actual reconfiguration process took only 18 seconds. The results of the experiments are summarized in Table 1.

Table 1 – Time required for software reconfiguration

Reconfiguration Component	**Location**	**Time (seconds)**
Monitor	OpenBrick	10
GCM and Controller	Base Station	10
Configurator	OpenBrick	8
Total Time		28
Total Reconfiguration Time		18

8.6 Summary

The experiments carried out for evaluation of the software reconfiguration architecture produced satisfactory results. They adequately demonstrated the need for software reconfiguration in sensor network applications. Our software reconfiguration architecture performed the dynamic software reconfiguration in just 28 seconds. Though this number is dependent on the number of components that will be ultimately switched and rewired, it is still a very small number when compared with the amount of effort and time needed if this activity is performed manually. The software infrastructure allows the selective switching of components. This feature of the infrastructure allows the dynamic replacement of some components with others without affecting the entire application.

CHAPTER IX

DYNAMIC SOFTWARE RECONFIGURATION IN MOTES BASED SENSOR NETWORKS

This approach for dynamic reconfiguration in sensor networks was also demonstrated using results obtained from simulations of Aislemonitor on the Berkeley MICA motes platform using TOSSIM [12] [16]. Reconfiguration was performed by a controller that was executed on a base station. The approach for software reconfiguration was based on Model Integrated Computing. The authors in [16] presented a modeling paradigm for TinyOS applications that supported the representation of alternative implementations of the same components along with explicit representation of constraints. The design space exploration tool (DESERT) was then used to evaluate the constraints based on measurements of the power available at each node and select an appropriate configuration. Once the next configuration was selected, the reconfiguration process involved stopping, rewiring and restarting the application components at the sensor nodes.

9.1 Platform Description

The work in [16] targeted wireless sensor networks that were based on energy and resource constrained devices. One of the most widely used platforms for researching wireless sensor networks with limited resources is the Berkeley MICA motes (described in the chapter 2) [11]. The MICA mote has a 4 MHz microcontroller, 4 KB of RAM, 128 KB of flash memory, 916 MHz wireless radio transceiver (19.2 Kbps transfer rate, 200

feet range) and is powered by two AA batteries and runs the TinyOS operating system, an open source, event driven and modular OS designed to be used with networked sensors. Daughtercards with various sensors and actuators are available, including photo, temperature, humidity, infra-red and barometric pressure sensors, accelerometers, magnetometers, and microphones, and sounders [3].

9.2 Architecture for Software Reconfiguration

The architecture for software reconfiguration proposed in this thesis was prototyped in [16] during the work on Berkeley motes. The architecture included (1) GRATIS [47] and (2) GRATISPlus [16], the modeling paradigms for representing TinyOS based applications in the Generic Modeling Environment (GME), (3) the design space exploration tool DESERT and (4) the Global Constraint Monitor (GCM). All these programs were deployed on the base station. Individual motes executed the (6) Monitor and (7) Reconfigurator components, in addition to the application components.

GRATIS is a modeling paradigm developed to model TinyOS applications. The graphical representation provides a solid and intuitive interface for designing and maintaining complex applications. GRATIS facilitates the visual modeling of *interfaces* and *modules* in *configurations* of TinyOS applications, and generates the textual representation of corresponding configuration files automatically. The environment also includes a mapping from the existing large code base of the system components included in the TinyOS distribution to the graphical environment. Using GRATIS, an application developer is able to model a new TinyOS application visually and then produce configuration files automatically using the GRATIS interpreter.

Using GRATISPlus, the user is able to model more than one design (alternative software components - *modules)* of the same TinyOS application, in a very compact and scalable representation, along with the constraints that are evaluated to generate valid configurations. In addition to the *interfaces*, *modules* and *configurations* represented in GRATIS models, GRATISPlus introduces an additional component called *group*, to allow modeling of alternative implementation of software components. A group typically contains more than one *module* representing alternative implementations of logic or algorithms. GRATISplus also allows the modeling of constraints using *condition* modeling objects. These constraints are expressed in the 'statement' attribute of a condition object in the Object Constraint Language (OCL). The design of SNRAMoLA is similar to the design of GRATISPlus in terms of applicability. The modules modeled in the group objects in GRATISPlus have an attribute for storing power values. These values are updated by the GCM during reconfiguration and evaluated by DESERT to generate valid configurations.

During reconfiguration, the GRATISPlus to DESERT interpreter converted the GRATISPlus models of the TinyOS application into a format compatible with DESERT. DESERT evaluated the constraints over the models and generated valid GRATIS configurations. The DESERT to GRATIS interpreter converted DESERT output to valid GRATIS models. The GRATIS interpreter converted the GRATIS models of the TinyOS application to actual configuration files.

The Monitor component executing on individual motes was responsible for measuring the local QoS parameters and communicating them to the base station. The

Reconfigurator was responsible for performing the necessary local application changes upon notification from the base station.

9.3 Case Study

The Aislemonitor application (explained in Chapter 6) was implemented on the TinyOS platform. Its purpose was the same, to track people walking in an aisle. A sensor network composed of eight motes was simulated in TOSSIM [12]. The application was deployed on all the motes. The tracking application was used to demonstrate the reconfiguration capabilities of the proposed architecture. The application was modeled in GRATISPlus. Different versions of the application were executed on the simulated sensor network and in TOSSIM. The data required for the simulation was generated using Matlab and provided to each mote through a text file.

The initial configuration of the Aislemonitor application included components that were able to detect the position of the person within range of the host motes. The data generated using Matlab was fed to the application and readings were recorded. We tested the reconfiguration architecture by switching off an intermediate mote, thus requiring its neighbor to perform software reconfiguration to predict the position of the people in the range of the disabled mote.

9.4 Performance Evaluation

The authors [16] tested the performance of the tracking application with and without reconfiguration for four test cases. Each test case included data for three people walking in one direction with varying speeds. The data was generated for 30 seconds

over 65 feet of the aisle assuming eight uniformly spaced motes. One of the motes was shut down between 8 to 9 seconds after the start of the simulation, thus simulating a low power condition. Using Aislemonitor #1 it was not possible to detect any people in the field of view of the affected mote and we had to switch to Aislemonitor #2. Two types of errors were encountered: (i) *People Missed*: As one of the motes shut down, people present in its field of view were not detected, and (ii) *People double counted*: The tracking algorithm continued to estimate a person in the field of the affected mote even after he/she had crossed to a neighboring mote.

Mote 2 was shut down in each of the test cases. Figure 24 shows the errors that occurred computed by summing up "people missed" and "people double counted" over the simulation interval. Aislemonitor #1 encountered 27 total errors and all of these were due to missed detection of people. Aislemonitor #2 reduced the "people missed" errors to 10. However, it generated 4 errors due to "people double counted". Overall, without reconfiguration we had 7.5% errors while with reconfiguration 3.8% errors.

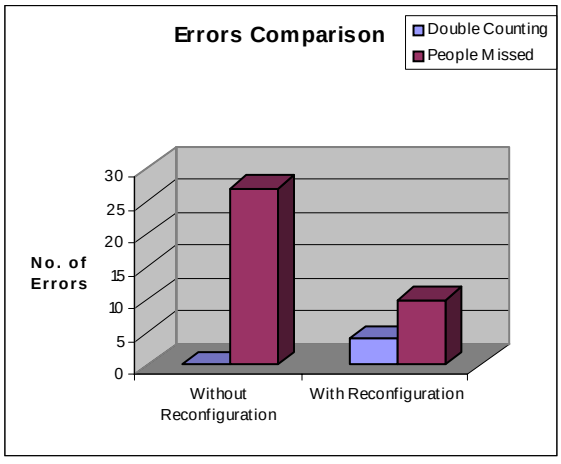

Figure 24 - Distribution of Errors in Aislemonitor #1 and Aislemonitor #2

CHAPTER X

CONCLUSIONS AND FUTURE WORK

10.1 Conclusions

This thesis presented an approach for dynamic software reconfiguration in sensor networks. Our approach requires monitoring the system requirements expressed as formal constraints. These constraints apply over critical parameters expressed as attributes in visual models of the application. The attributes are updated by dedicated monitoring components, which gather data from the entire network. A change in the attribute values beyond set thresholds drives the reconfiguration process that takes place at a base station that can communicate to all the sensor nodes.

The proposed approach has been demonstrated using results obtained from a simple one-dimensional tracking problem. All parts of the software reconfiguration infrastructure have been evaluated. The need for a software reconfiguration architecture for sensor network applications is apparent from the results of even a simple tracking algorithm. The time required for a particular OpenBrick device to reconfigure its components is around 18 to 28 seconds, which is very less when compared to the cost of manually stopping and restarting the application with the correct components. In sensor network applications running over a long duration, the ability to reconfigure the software components, resulting in a change in the behavior of the application, in response to external stimuli, in such a short time is of special significance. The automatic reconfiguration of components expressed in a user friendly modeling environment on a base station in response to changing operating conditions in the field, and the

communication of the new configuration to the individual nodes to reconfigure their application is a very attractive solution to the problem of managing a large sensor network.

10.2 Future Work

While the modeling, design-space exploration and reconfiguration, communication and the reconfiguration infrastructure tools are implemented and tested, work needs to be done to construct methods to enumerate and capture more critical QoS parameters that represent the communication and computation resources of the sensor network (for example, bandwidth utilized, and number of executing processes). Reconfiguration is triggered by a change in these values. The monitor components responsible for monitoring the health of the system are integrated with the application components in the current reconfiguration architecture. A more generic architecture for the development and implementation of the monitor components will significantly improve the capture of more QoS critical parameters from the field. A similar improvement can be done with the GCM executing on the base station. A strong software architecture for updating the parameters in the SNRAMoLa models will simplify the design of the applications and standardize the reconfiguration process.

Reconfiguration is performed when the QoS attributes exceed a set threshold. These thresholds may be different for different application domains. A standard method for capturing the desired levels of QoS parameters will simplify the application of software reconfiguration in various domains.

APPENDIX

A. SOFTWARE RECONFIGURAITON ARCHITECTURE SETUP

This section contains setup details of the software reconfiguration architecture. The software reconfiguration architecture can be divided into two parts based on the location of the executing components: (1) Base Station and (2) OpenBrick device.

A.1 RECONFIGURATION COMPONENTS ON BASE STATION

The software reconfiguration infrastructure components on the base station are: (1) SenNet.conf file (2) SNRAMoLa models of the application, (3) Controller Program (4) SNRAMoLa to DESERT interpreter, (5) DESERT, (6) DESERT to Configurator interpreter, and (7) Global Constraint Monitor (GCM).

A.1.1 SenNet.conf File

SenNet.conf is the configuration file of the reconfiguration architecture on the base station. It contains (1) Packet Forwarder IP and Port information (used by Controller), (2) SNRAMoLa model name and path information (used by Controller and GCM), (2) DESERT executable path and directory information (used by Controller), (3) SNRAMoLa to DESERT interpreter executable path and directory information (used by Controller), (4) DESERT to Configurator interpreter executable path and directory information (used by Controller), (5) Controller program executable path and directory information (used by GCM), and (6) Host device IP address and listening Port (used by GCM). This file is currently located in the ‘C:\Documents and Settings\kogekasv\Thesis\MetaModel’ folder on the base station. This folder also

contains the DesertInput, DesertOutput, GCM, and Talker folders, which contain the SNRAMoLa to DESERT, DESERT to Configurator, GCM, and Controller programs respectively. The directory structure for the base station components should be maintained by keeping the 'SenNet.conf' file along with the other mentioned folders in the one folder at the same level in the hierarchy.

A.1.2 SNRAMoLa Models of the application

SNRAMoLa models of the application are stored as '.mga' files. The model of the Aislemonitor application is stored in the 'SenMetaData_test.mga' file located in the 'C:\SenMetaData\' folder on the base station. These models are created in GME using the SenMeta modeling paradigm. Before deploying the application on the sensor network, the name of the model and its path is recorded in the 'SenNet.conf' configuration file.

A.1.3 Controller Program

The controller program is invoked by the GCM upon receipt of messages from the Monitors executing on individual OpenBricks. Its path is specified in the 'SenNet.conf' configuration file. The controller program is located in the 'C:\Documents and Settings\kogekasv\Thesis\MetaModel\Talker\' folder on the base station. The Controller program reads from the 'SenNet.conf' configuration file and invokes SNRAMoLa to DESERT, DESERT and DESERT to Configurator interpreters sequentially. It then sends the newly created configuration files to individual OpenBrick devices via the Packet Forwarder as UPD packets. The Controller uses the IP address and port data read from the SenNet.conf file to address the packets.

A.1.4 SNRAMoLa to DESERT Interpreter

The SNRAMoLa to DESERT interpreter converts the application models to a format compatible with DESERT. The input to DESERT is stored in the 'DesertInput.xml' file. The Controller program invokes the interpreter and passes the name of the model along with the target directory as command line parameters. The output XML file stored in the target directory, which is the same folder ('C:\SenMetaData\') as the application model ('SenMeta_test.mga'). The SNRAMoLa to DESERT Interpreter is stored in the 'C:\Documents and Settings\kogekasv\Thesis\MetaModel\DesertInput\' folder on the base station.

A.1.5 DESERT

The Controller program invokes DESERT using the information in the SenNet.conf file and passes the file 'DesertInput.xml' to it as a command line parameter. This file contains the SNRAMoLa models of the application in a form compatible with DESERT. DESERT performs design space exploration and stores its output in the 'DesertInput_back.xml' file in the same folder as 'DesertInput.xml'. In the present application, this file is stored in the 'C:\SenMetaData\' folder. DESERT is invoked using the 'DesertTool.exe' executable stored in the 'C:\Desert\' folder on the base station.

A.1.6 DESERT to Configurator Interpreter

The Controller program invokes this interpreter using the information in the SenNet.conf file and passes the files 'DesertOutput.xml' and 'SenMeta_test.mga' as command line parameters. The interpreter uses these two files to generate the

configuration files for each OpenBrick device. These configuration files are also stored in the same folder as the application model (‘C:\SenMetaData’). The DESERT to Configurator interpreter is stored in the ‘C:\Documents and Settings\kogekasv\Thesis\MetaModel\DesertOutput\’ folder on the base station.

A.1.7 Global Constraint Monitor (GCM)

The GCM is stored in the ‘C:\Documents and Settings\kogekasv\Thesis\MetaModel\GCM\’ folder on the base station. The GCM reads from the SenNet.conf file and uses the IP address information stored in it to open a socket for listening for incoming messages from OpenBrick devices. Upon receipt of a message, it updates the critical QoS attributes in the SNRAMoLa models of the application and then invokes the Controller program and triggers reconfiguration on the base station.

A.2 RECONFIGURATION COMPONENTS ON OPENBRICK DEVICES

The software reconfiguration infrastructure components on individual OpenBrick devices are: (1) Configurator, (2) Communicator, and (3) Monitor.

A.2.1 Configurator

The Configurator component is installed in the '/root/sachin/aislemonitor/configurator' folder on all OpenBrick devices. Information about Packet Forwarder IP addresses and ports, Configurator IP addresses and ports is stored in the 'receive.conf' file in the same folder. The Configurator reads this file and opens up a socket to listen for incoming reconfiguration commands from the base station. If the host OpenBrick device is connected to the base station through a wired connection, the Configurator also sets up the Packet Forwarder on the device. The configurator is invoked by using the '/root/sachin/aislemonitor/configurator/configurator' command.

A.2.2 Communicator

The Communicator component is stored in the '/root/sachin/aislemonitor/configurator' folder on each OpenBrick device. The 'communicator.h' file, which implements the Communicator component, is required to be included by each application process. Using the Communicator component, processes establish shared memory connections with the Configurator to receive reconfiguration commands. The Communicator is initialized by the process using the 'communicator_init()' method.

A.2.3 Monitor

The Monitor used in our test-bed is integrated with the Aislemonitor application. It has not been implemented as a separate component.

The reconfiguration architecture is deployed by:

1. Starting the Configurator components on all the OpenBrick devices using the '/root/sachin/aislemonitor/configurator/configurator' command.
2. Starting GCM on the Base Station.
3. Invoking the Talker Program manually to generate the first set of configuration files.

Once started, the dynamic software reconfiguration architecture performs software reconfiguration automatically as and when it receives failure notifications from the Monitor components and continues to execute till it is stopped manually by sending the "STOPLISTENING" message from the base station.

B. AISLEMONITOR APPLICATION SETUP

This section describes the setup of the Aislemonitor Application deployed on each OpenBrick device. The Aislemonitor application is composed of the (1) ImageSensor, (2) Receiver, (3) Estimator1, (4) Estimator2, (5) DataCollector1, and (4) DataCollector2 components. Each component is implemented as a separate program.

B.1 ImageSensor

The ImageSensor component is installed in the '/usr/local/motion3/' folder on each OpenBrick device. The Configurator executes it using the command: '/usr/local/motion3/motion *memoryId*:*memoryName'* (explained in Chapter 7). Various control variables like the frame-rate are set in the 'motion.conf' file located in the same folder. The ImageSensor reads from this file during initialization.

B.2 Receiver

The Receiver program is installed in the '/root/sachin/aislemonitor/receiver/' folder on each OpenBrick device. The Configurator executes it using the command: '/root/sachin/aislemonitor/receiver/receiver *memoryId*:*memoryName*' Information about the IP addresses and ports of neighboring OpenBrick devices as well as listening ports on the host OpenBrick device is stored in the receive.conf file located in the same folder. The Receiver program reads this file during initialization.

B.3 Estimator1

The Estimator1 program is installed in the '/root/sachin/aislemonitor/Estimator3/' folder on each OpenBrick device. The Configurator executes it using the command: '/root/sachin/aislemonitor/Estimator3/estimator *memoryId*:*memoryName*.'

B.4 Estimator2

The Estimator2 program is installed in the '/root/sachin/aislemonitor/Estimator4/' folder on each OpenBrick device. The Configurator executes it using the command: '/root/sachin/aislemonitor/Estimator4/estimator *memoryId*:*memoryName*.'

B.5 DataCollector1

The DataCollector1 program is installed in the '/root/sachin/aislemonitor/application3/' folder on each OpenBrick device. Its executable is called 'application'. The program reads from the 'aislemonitor.conf' file stored in the '/root/sachin/aislemonitor/application/' folder. This file stores information about the left and right range limits of the host OpenBrick device and is used by the DataCollector, ImageSensor and Estimator components.

B.5 DataCollector2

The DataCollector2 program is installed in the '/root/sachin/aislemonitor/application4/' folder on each OpenBrick device. It is executed using the command '/root/sachin/aislemonitor/application4/application

memoryId:*memoryName*' by the Configurator. The program reads from the 'aislemonitor.conf' file stored in the '/root/sachin/aislemonitor/application/' folder.

All these programs include the 'communicator.h' file located in the '/root/sachin/aislemonitor/configurator/' folder and invoke the 'communicator_init()' function during initialization.

REFERENCES

[1] http://www.antd.nist.gov/wahn_ssn.shtml, Project: Wireless Ad hoc Networks, Advanced Network Technologies Division, National Institute of Standards and Technology, Gaithersburg, Maryland.

[2] Cerpa A., Elson J., Estrin D., Girod L., Hamilton M. and Zhao J., "Habitat Monitoring: Application driver for wireless communications technology," Proceedings of the ACM SIGCOMM Workshop on Data Communications, Latin America and the Caribbean, April 2001.

[3] Mainwaring A., Polastre J., Szewczyk R., Culler D. and Anderson J., "Wireless Sensor Networks for Habitat Monitoring," ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), Atlanta, GA, September 2002.

[4] Biagioni E., and Bridges K., "The Application of Remote Sensor Technology to assist the recovery of rare and endangered species," Special issue on Distributed Sensor Networks for the International Journal of High Performance Computing Applications, Vol. 16, N. 3, August 2002.

[5] Yang H. and Sikdar B., "A Protocol for Tracking Mobile Targets using Sensor Networks," Proceedings of IEEE Workshop on Sensor Network Protocols and Applications, 2003.

[6] http://www.citris.berkeley.edu/applications/disaster_response/smartbuildings.html, Project: Smart Buildings, CITRIS, Berkeley, CA.

[7] http://www-white.media.mit.edu/vismod/demos/smartroom/, Project: Smart Rooms, Massachusetts Institute of Technology.

[8] Simon G. et al., "Shooter Localization in Urban Environments," Information Processing in Sensor Networks (Submitted), April 2004.

[9] Srivastava M. B., Muntz R. R. and Potkonjak M., "Smart Kindergarten: Sensorbased Wireless Networks for Smart Developmental Problem-solving Enviroments," Mobile Computing and Networking, pages 132.138, 2001.

[10] Sztipanovits J., Karsai G.: "Model-Integrated Computing," IEEE Computer, April 1997.

[11] Hill J. and Culler D., "Mica: A Wireless Platform for Deeply Embedded Networks," IEEE Micro, vol. 22(6), Nov/Dec 2002, pp 12-24.

[12] Levis P., Lee N., Welsh M., Woo, and Culler D., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," ACM SensSys 2003, Nov. 2003.

[13] Estrin D., Girod L., Pottie G., Srivastava M., "Instrumenting the world with Wireless Sensor Networks," International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001), Salt Lake City, Utah, May 2001

[14] Bachrach J., Laddaga R., Robertson P., Salib M., Shrobe H., "Sensor Nets for Military Operations in Urban Terrain."

[15] Hill J., Szewczyk R., Woo A., Hollar S., Culler D. and Pister K., "System Architecture Direction for Networked Sensors," ACM SIGPLAN Notices, vol. 35(11), Nov 2000, pp93-104.

[16] Kogekar S., Neema S., Eames B., Koutsoukos X., Ledeczi A., and Maroti M.: "Constraint-Guided Dynamic Reconfiguration in Sensor Networks," IPSN '04, April 26-27, 2004 Berkeley, California, USA.

[17] Royer E.M. and Toh C-K, "A Review of Current Routing Protocols for Ad hoc mobile wireless networks," IEEE Personal Communications, April 1999.

[18] Perkins C. E. and Royer E.M., "Ad-hoc On-Demand Distance Vector Routing," Proceedings from 2nd IEEE Workshop on Mobile Computing Systems And Applications, Feb 1999, pp. 90-100.

[19] Davis J., Scott J., Sztipanovits J., Karsai G., Martinez M.: "Integrated Analysis Environment for High Impact Systems," Proceedings of the Engineering of Computer Based Systems, Jerusalem, Israel, April 1998.

[20] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi A.: Metamodeling – "Rapid Design and Evolution of Domain-Specific modeling Environments," Proceedings of the IEEE Conference and Workshop on Engineering of Computer Based Systems, April 1999.

[21] Cheng S-W, Garlan D., Schmerl B., Sousa J. P., Spitznagel B., Steenkiste P., Hu N., "Software Architecture Based Adaptation for Pervasive Systems," International Conference on Architecture of Computing Systems Trends in Network and Pervasive Computing, Karlsruhe, Germany, April 8-11, 2002

[22] Castaldi M., Carzaniga A., Inverardi P., Wolf A. L., "A Lightweight Infrastructure for Reconfiguring Applications," In Proceedings of 11th Software Configuration Management Workshop, Portland, Oregon, USA, May, 2003

[23] Derk M.D., DeBrunner L. S., "Reconfiguration for Fault Tolerance using Graph Grammars," ACM Transactions on Computer Systems (TCOS) Volume 16, Issue 1, February 1998, pp 41-54.

[24] Mitchell D., Naguib H., Coulouris G., Kindberg T., “A QoS Support Framework for Dynamically reconfigurable Multimedia Applications,” Proceedings of the IFIP WG 6.1 International Working Conference on Distrubted Applications and Interoperable Systems II, p.17-30, June 28-July 01, 1999

[25] Villasenor J., Schoner B., Chia K.N., Zapata C., Kim H.J., Jones C., Lansing S., and Mangione-Smith B., “Configurable Computing Solutions for Automatic Target Recognition,” Proceedings of the IEEE Symposium of FPGAs for Custom Computing Machines, pp. 70-79, Napa California, Apr. 17-19, 1996

[26] Ghiasi S., Sarrafzadeh M., “Optimal Reconfiguration Sequence Management,” Asia South Pacific Design Automation Conference (ASPDAC), pp. 359-365, January 2003.

[27] Allen R., Douence R. and Garlan D., “Specifying and Analyzing Dynamic Software Architectures,” Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal, March 1998.

[28] Lee E. A., Messerschmitt D.G., “Synchronous Data Flow”, Proceedings of the IEEE Vol. 75, No. 9 September 1987.

[29] Guibas L., “Sensing, Tracking and Reasoning with Relations,” IEEE Signal Processing Magazine, vol. 19, no. 2, pp. 73-85, March 2002.

[30] Welch G., Bishop G., “An Introduction to the Kalman Filter,” Report: TR95-041, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1995.

[31] Maybeck P. S., “Stochastic Models, Estimation, and Control,” Book: Chapter 1, Academic Press.

[32] http://www.openbrick.org

[33] http://w3.antd.nist.gov/wctg/aodv_kernel/

[34] http://motion.sourceforge.net/

[35] http://www.thedirks.org/v4l2/v4l2fmt.htm

[36] Neema S., Sztipanovits J., Karsai G., Butts K.: “Constraint-Based Design-Space Exploration and Model Synthesis.” LNCS 2855, pp 290-305, Sept 2003.

[37] Ledeczi A., Bakay A., Maroti M., Volgyesi P., Nordstrom G., Sprinkle J., Karsai G., “Composing Domain-Specific Design Environments,” Computer, pp. 44-51, November 2001.

[38] Warner J. B., Kleppe A. G.: “The Object Constraint Language: Precise Modeling With Uml,” Addison-Wesley, 1999.

[39] Bryant R.: “Symbolic Boolean manipulation with ordered binary-decision diagrams,” ACM Computing Surveys, Volume 24, Issue 3, September 1992.

[40] Neema S.: “Design Space Representation and Management for Embedded Systems Synthesis,” Technical Report, ISIS -01-203, February 2001.

[41] http://www.xbow.com/Products/productsdetails.aspx?sid=85

[42] Lee E.A. and Parks T.M., “Dataflow Process Networks”, Proceedings of IEEE, 83(5): 773-801, May 1995.

[43] Bulusu N., Heidemann J., Estrin D., “GPS-less Low Cost Outdoor Localization for Very Small Devices. IEEE Personal Communications Magazine,” IEEE Personal Communications Magazine, Vol. 7, No. 5, pp 28-34. October, 2000.

[44] Bulusu N., Bychkovskiy V., Estrin D., Heidemann J., “Scalable, Ad Hoc Deployable RF-based Localization,” Proceedings of the Grace Hopper Celebration of Women in Computing Conference 2002, Vancouver, British Columbia, Canada. October 2002.

[45] Savarese C., Rabaey J. M., Beutel J., “Locationing in Distributed Ad-Hoc Wireless Sensor Networks,” Proceedings of the ICASSP, May 2001.

[46] Sztipanovits J., Karsai G., Franke H., “Model-Integrated Program Synthesis Environment,” ECBS, March 11-15, 1996, Friedrichshafen, Germany.

[47] Volgyesi P., Ledeczi A., “Component-Based Development of Networked Embedded Applications,” EUROMICRO ‘02, September 4-6, 2002, Dortmund, Germany.

[48] http://www.nist.gov/dads/HTML/modelOfComputation.html