Constraint-Guided Dynamic Reconfiguration in Sensor Networks

Sachin Kogekar, Sandeep Neema, Brandon Eames,
Xenofon Koutsoukos, Akos Ledeczi, and Miklos Maroti

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN 37235
+1 (615) 343 7472

{sachin.kogekar, sandeep.neema, brandon.eames, xenofon.koutsoukos, akos.ledeczi, miklos.maroti}@vanderbilt.edu

ABSTRACT
This paper presents an approach for dynamic software reconfiguration in sensor networks. Our approach utilizes explicit models of the design space of the embedded application. The design space is captured by formally modeling all the software components, their interfaces, and their composition. System requirements are expressed as formal constraints on QoS parameters that are measured at runtime. Reconfiguration is performed by transitioning from one point of the operation space to another based on the constraints. We demonstrate our approach using simulation results for a simple sensor network that performs one-dimensional tracking.

Categories and Subject Descriptors
C.3 [**Computer Systems Organization**]: Special Purpose and Application-based Systems – *Real time and embedded systems.*

General Terms
Algorithms, Management, Design

Keywords
Runtime/Dynamic Software Reconfiguration, Design Space Exploration

1. INTRODUCTION

Reconfiguration and self-adaptation are vital capabilities of sensor networks that are required to operate in dynamic environments that impose varying functional and performance requirements.

IPSN'04, April 26–27, 2004, Berkeley, California, USA.

Dynamically adaptive software comprises of tasks that detect internal and external changes to the system, reflect on the event occurrences, and adapt to the new conditions. Ad hoc wireless sensor networks, in particular, must be designed with adaptation capabilities that enable them to handle a multitude of operating conditions. Reconfiguration in such systems presents significant challenges because of the severe constraints in energy, computation, and communication resources. 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. This is in contrast to the current state-of-the-art in that it does not allow embedded software to evolve at runtime. Conventional practices for embedded software development rely on iterating single-point designs. This, in effect, implies elimination of component and system design alternatives in the early stages of the development process. Such elimination leads to suboptimal and inflexible design that is difficult and expensive to deploy and maintain.

We present an approach for building self-adaptive sensor networks based on Model-Integrated Computing [1]. Our approach is utilizing explicit models of the *design space* of the embedded application. The design space is captured by formally modeling all the software components, their interfaces, and their composition. 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 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.

The key question in reconfiguration of sensor networks is how to decide what the new configuration should be. This 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. We propose an efficient approach for performing this search 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 con-

figuration that satisfies all the constraints is found, the reconfiguration can be accomplished by online software synthesis targeting either an interpreted language (e.g. Java) or a command interface.

We demonstrate our approach using simulation results for a simple sensor network that performs one-dimensional tracking based on the Berkeley MICA motes. Reconfiguration is performed by a controller that runs on a base station that is capable of monitoring the constraints. We present a modeling paradigm for TinyOS applications that supports alternative implementations of the same components and explicit representation of constraints. Our design space exploration tool is then used to evaluate the constraints based on measurements of the power available at each node – the primary resource constraint in this application – and selects an appropriate configuration. Once the next configuration has been selected, the reconfiguration process involves stopping, rewiring and restarting the application components at the sensor nodes.

Software reconfiguration methods have been widely applied in areas such as hardware/software co design and multimedia applications [18]. The applicability of such methods to sensor networks is a very challenging problem due to resource constraints. Combinatorial search techniques and heuristics are applied to the problems of design partitioning, architecture mapping, scheduling and synthesis. A methodology for system synthesis is provided in [2], which utilizes evolutionary algorithms to optimize the partitioning and allocation of a dataflow-based application specification onto a heterogeneous processing network. Design space exploration techniques have been utilized for integrating simulation engines together with a design-space exploration tool to facilitate embedded systems design [3], for evaluating and optimizing low-level partitioning and mapping decisions for embedded systems which target morphable platforms [4], and for partitioning designs onto a heterogeneous network of DSPs and FPGAs [5]. A design space exploration tool to aid in the development of embedded software for automotive applications has been developed in [6]. A related approach for online reactive constraint solving motivated by the needs of resource constrained embedded systems such as printers has been presented in [7].

Active software [8] is a related research direction motivated by requirements such as handling increased functional complexity, providing substantially increased robustness, and providing autonomy. An overarching theme of this approach is that software must take active responsibility for its own robustness and the management of its own complexity, and in order to do so, software must incorporate representation of its goals, methods, alternatives, and environment. Candidate technologies for active software include self-adaptive software, negotiated coordination, tolerant software and physically grounded software. In our view, our approach falls under the umbrella of self-adaptive software. In [8], evaluation is identified as the hardest and most important problem for self-adaptive software. We argue that performance evaluation can be managed with a constraint-based approach. Evaluation of constraints parameterized with the operational parameters gives a clear and quantifiable understanding of the system performance.

The remainder of the paper is organized as follows. Section 2 presents the proposed architecture for reconfiguration of sensor networks. The constraint-based design space exploration approach is described in Section 3. In Section 4, the approach is applied to sensor networks based on a graphical modeling paradigm that supports alternatives and constraints. A case study of a tracking application is presented in Section 5.

2. SOFTWARE RECONFIGURATION IN SENSOR NETWORKS

2.1 Platform Description

Our work targets wireless sensor networks that are 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 [9]. 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. Daughter cards with various sensors and actuators are available, including photo, temperature, humidity, infrared and barometric pressure sensors, accelerometers, magnetometers, and microphones, and sounders [10].

The Berkeley MICA motes run the TinyOS operating system [9], 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 [11]. 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. The motes themselves 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 motes that execute them. We assume that the motes do have enough program memory to store alternative components. This assumption is reasonable; it is typically the data memory that is the limiting factor for MICA applications. Our reconfiguration approach is detailed in the next section.

2.2 Architecture for Software Reconfiguration

We have prototyped an architecture for software reconfiguration shown in Fig.1. In addition to the application, each mote in the

network contains a Monitor and a Reconfigurator component. The Monitor is responsible for measuring the local QoS parameters and communicating them to the base station. The Reconfigurator is responsible for performing the necessary local application changes upon notification from the base station. The base station contains the following components: (i) the Global Constraint Monitor (GCM), (ii) the GRATISPlus modeling environment that supports alternatives and explicit representation of constraints, (iii) the design space exploration tool DESERT for selecting the desired configuration, and (iv) the GRATIS modeling environment for generating the new TinyOS configuration.

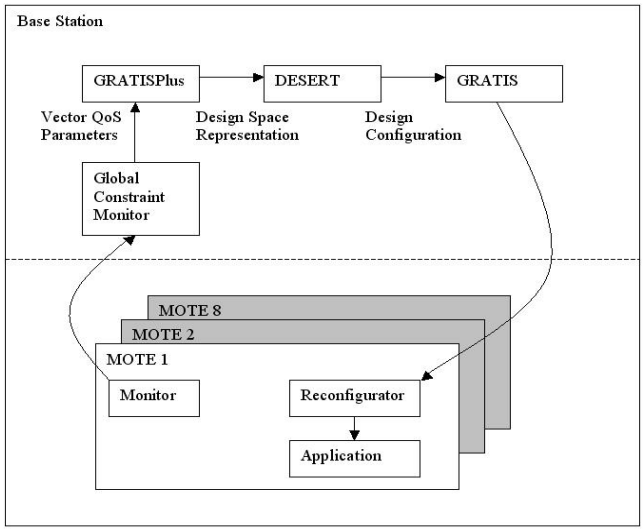

Figure 1. Software architecture for reconfiguration

The Monitor gathers information about certain critical QoS parameters such as available 'Power' at each mote. The Monitor sends the parameters to the GCM at the base station. The GCM has a database of parameters, the dependent constraints and associated thresholds. If the parameter changes are above the thresholds in the database, the GCM updates the constraints in the GRATISPlus modeling environment. The GRATISPlus to DESERT mapping interpreter generates a representation of the GRATISPlus model and constraints to be used as the input to DESERT. DESERT evaluates these constraints and generates single-point designs that are valid with respect to the updated constraints. This is mapped into a new GRATIS model which then generates the new TinyOS configuration. The old and new configuration files are then analyzed by another tool that generates the elementary reconfiguration commands specifying which components need to be stopped, rewired and (re)started. The list of commands are sent to the Reconfigurator on each mote. The Reconfigurator executes the command list and restarts the application. Hence, the application has been dynamically reconfigured to adapt to changes in QoS parameters.

We realize that this approach of keeping global parameter knowledge in a centralized database has limitations with respect to scalability and introduces a single point of failure. However, with the current limitations of the sensor nodes and the smaller size sensor networks that we are prototyping our approach with this organization suffices. We are planning to extend this towards hierarchical database organization as scale up the sensor networks

2.3 Reconfigurator

TinyOS applications are statically compiled. Neither the memory frame of components, nor the wiring between components (compiled as function calls) can be changed at run-time. Therefore, supporting software reconfiguration within the TinyOS architecture requires new components that assist during reconfiguration and perform the dynamic wiring of components. At least two issues need to be addressed. First, we have to introduce new and remove old components from the graph of running components during software reconfiguration. Second, we need to dynamically change the wiring of components. A module, called the *Reconfigurator*, orchestrates the reconfiguration.

We say that a component is *reconfigurable*, if it is not included in all configuration graphs. We require that each reconfigurable component implements the StdControl interface, which is the standard TinyOS interface for starting and stopping a component. During software reconfiguration, the Reconfigurator first stops all components that need to be disabled. Note, that components cannot veto this decision. Stopping a component may require additional tasks that cannot be performed immediately. Nevertheless, the stopped component must finish its de-initialization using a fixed number of tasks, which cannot post further tasks, and reach a state where it does not participate in any further communication. This rule allows the Reconfigurator to wait till all stopped components are completely de-initialized.

The edges of the configuration graphs represent interfaces links connecting the provider to the user of the interface. We say that a link is *reconfigurable* if it does not appear in all configuration graphs. Each component connected to a reconfigurable link must determine the target component dynamically when calling a command. This is best achieved by replacing the reconfigurable links of components by links to specialized *switching components* that make the dynamic binding. The implementation of these switching components depends on the involved interface and may include additional logic for keeping the contract of the interface during the switch. For example, the SendMsg interface assumes that each 'send' command is followed by a 'sendDone' event. When the switching component makes the switch at a point when the command has been called but the event has not arrived, it must respect the implicit contract by firing the 'sendDone' to keep the user of the interface synchronized. Each switching component must implement the 'rewire' command that takes a configuration number, and based on the current configuration number route the commands and events using precompiled switch and case statements.

After the reconfigurator stops all components that are not used in the new configuration, it calls the rewire command on each switch component. Finally, it calls the start command of the StdControl interface on each component that need to be started in the new configuration.

2.4 The GRATIS Modeling Environment

Textual representation of component composition, especially when it is spread across multiple files is inherently error-prone. Even in the simplest scenario a more expressive representation of our components and the interconnections between them can help us avoid errors and help others understand the application. With more complex components and especially with hierarchical composition this becomes an absolute requirement. Model Integrated Computing [1] in general and the Generic Modeling Environment (GME) in particular can meet these challenges.

We have designed a modeling paradigm for GRATIS and configured GME [12] accordingly. The graphical representation provides a solid and intuitive interface for designing and maintaining complex applications. The constraint management capabilities of GME allow us to specify the necessary syntactical constraints for our components and for the usage of their interfaces. Our initial goal with GRATIS was to specify the interface and configuration information visually, and generate the corresponding textual representation automatically. Since all practical applications use system components from the TinyOS distribution, we also had to provide a mapping from the existing large code base to the graphical environment. Therefore, our interpreter not only generates text files from graphical models, but it is also capable of parsing existing files and building the corresponding GRATIS models from them. This way we can provide all TinyOS components as a library to the user of the GRATIS environment.

2.5 The GRATISPlus Modeling Environment

GRATISPlus is an extended version of GRATIS [12]. Using GRATISPlus, the user can model more than one implementation (alternative software components - viz. *modules)* of the same TinyOS application, in a very compact and scalable representation, along with the constraints that can be evaluated to generate valid configurations.

The core concepts in the GRATIS modeling paradigm include *interfaces*, *modules* and *configurations* representing TinyOS software components of the same name. The interactions between these components are modeled with *link* and *equate* connections. A link specifies that a component provides and another uses a certain interface. An equate connection specifies that the implementation of an interface is delegated to a subcomponent. GRATISPlus introduces an additional component called *group*, to allow modeling of alternative implementation of software components. A group can contain *modules*, *interfaces* and *equate* connections. It typically contains more than one *module* representing alternative implementations of logic or algorithms. GRATISplus also allows for modeling of constraints using *condition* modeling objects that are contained in *configurations.* The 'statement' attribute of a condition object contains the actual constraint expressed in the Object Constraint Language (OCL) [17]. E.g. self.power <= .05. The GRATISPlus meta-model is shown in Fig.2.

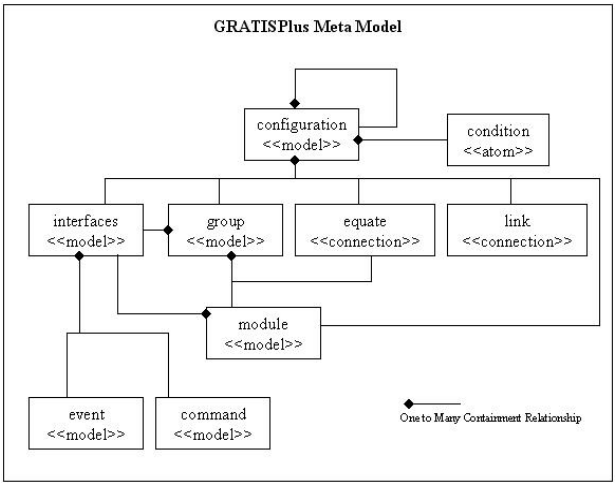

Figure 2. GRATISPlus Meta-Model

3. CONSTRAINT BASED DESIGN EXPLORATION

In this section, we describe the constraint based design space exploration implemented in the DESERT tool. Design space exploration in DESERT 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. In the rest of this section, we formalize the representation of design spaces in DESERT, and then we briefly review the encoding and pruning of design spaces.

3.1 Design Space Representation

Before we formalize the notation, we introduce the modeling language used by DESERT to represent design spaces. Fig. 3 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 the Element is composed of all its children 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 the 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.

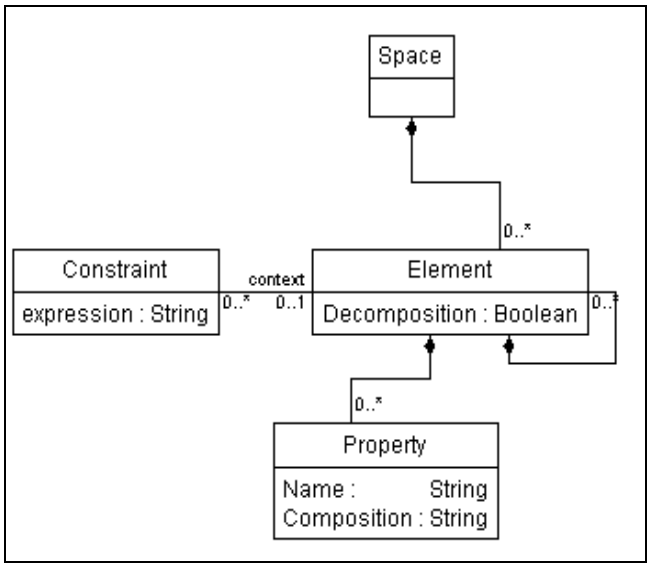

Figure 3. Meta-model of DESERT

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. Notice that 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 policy.

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), we use a variant of OCL for constraint specification.

We summarize this mechanism of structuring design spaces as hierarchically layered parameterized alternatives, and demonstrate its scalability in representing large design spaces with the following example: 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 we will show its formulation in the following expressions. In formulating a space we introduce the notion of a configuration. 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, we can define the set of possible instantiations $PS(l)$ of the leaf element l as:

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

Now, we can construct the set of configurations recursively, depending on element decomposition, as follows:

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

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:

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

3.2 Design Space Encoding and Pruning

Notice that since we are focusing on structural semantics of the design space and intend to compute with structural constraints, 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, we choose to perform the manipulation operations *symbolically*. Two problems had to be solved: 1) symbolic representation of design-spaces, and 2) symbolic representation of constraints.

If we restrict the parameters of model objects 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 [13]. These can then be symbolically manipulated with Ordered Binary

Decision Diagrams (OBDD-s) [13], 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 our encoding scheme have been described in [14].

We identify two basic categories of structural constraints that DESERT can compute efficiently. We discuss their symbolic representation here briefly.

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.

Note that the constraints that we consider here are different from temporal constraints, which assert temporal invariants over a dynamically evolving system. Temporal constraints can be valuable in pre-verifying temporal properties about an evolving system, however, temporal the verification is based on the assumption of a known/characterizable state-space of the system. The state-space of a sensor network on the other hand is extremely difficult to define and characterize. Therefore, we take a approach of asserting constraints that are temporal invariants in the sense that they define properties which should hold over different possible (non-deterministic) evolution paths of the system, however, we do not pre-verify these assertions, instead we check these assertion during system operation at each evolution step.

According to our experience, 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. We are experimenting with re-encoding the design-space after each pruning step, which usually results in a drastic decrease in the number of binary variables.

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 is to find a single design configuration (not all), which satisfies the selected design constraints. We currently experiment with various constraint solvers and languages, such as Oz [15] to develop solution for this approach.

4. MODELING OF ALTERNATIVES AND CONSTRAINTS

In this section, we formally define the design space for an application modeled in the GRATISPlus modeling language, which was introduced in section 2.5. We then show how this design space can be mapped onto DESERT, thereby allowing exploration and pruning of the space by applying the captured constraints.

A GRATISPlus model Gp is a tuple (T_g, C), where C is the set of GRATISPlus constraints, and T_g is a tree (N, E). The vertex set N of T_g is the set of software components (i.e. interfaces, modules, group, and configurations). The directed edge set E of T_g represents containment relation between hierarchical modeling elements.

$$\forall v_1, v_2 \in N \qquad (4)$$
$$v_2 \in children(v_1) \; iff \; \exists e = (v_1, v_2) \in E$$

where $children(n)$ is the set of objects contained in a an object n. We also define $Cfg \subset N$, $Grp \subset N$, and $Mod \subset N$ as disjoint set of objects of type *Configuration*s, *Group*s, and *Modules* respectively in a GRATISPlus model. As mentioned earlier, a configuration exhibits AND-decomposition semantics, while a group exhibits OR-decomposition semantics. GRATISPlus Modules are characterized with properties, over which the constraints are evaluated. An example of such a property is the power consumed by a particular module.

A 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)

We now discuss the mapping of GRATISPlus onto DESERT. The execution of such a mapping allows the DESERT tool to be used to explore and prune this design space. Under this mapping a GRATISPlus model maps onto a single DESERT *Space*, whose elements represent GRATISPlus components.

We define $Gp2Des : N \leftrightarrow S$ as a bijection, from objects in Gp to elements in DESERT, S being the set of DESERT elements. The following holds under this mapping:

$$\forall v_1, v_2 \in N \quad (5)$$
$$v_1 \in children(v_2) \Leftrightarrow Gp2Des(v_1) \in \chi(Gp2Des(v_2))$$

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

$$\forall de \in S \quad (6)$$
$$decompositiont(de) = \begin{cases} true & GpDes^{-1}(de) \in Cfg \\ false & GpDes^{-1}(de) \in Grp \end{cases}$$

The mapping of GRATISPlus constraints to DESERT is straightforward, and simply involves the mapping of the GRATISPlus constraint context onto its projection element under the $Gp2Des$ bijection. The properties of GRATISPlus modules are mapped to properties in the corresponding DESERT element, with the values appropriately associated. Once the mapping from GRATISPlus onto DESERT is complete for an application, DESERT prunes and explores the design space. The set of Design configurations that meet all constraints are returned, and the results are mapped back onto the GRATISPlus application. The mapping from DESERT back onto GRATISPlus (or equivalently GRATIS) is very simple. 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. This selection is mapped back onto a GRATIS representation by modifying the original GRATISPlus tree to replace each vertex of type Group with its corresponding child, per the binding contained in the configuration. The resulting GRATIS model has all design decisions resolved, and consists purely of vertices of type Module and Configuration. All Group nodes are resolved by the DESERT pruning process.

5. CASE STUDY: ONE-DIMENSIONAL TRACKING

5.1 Problem Description

The purpose of this TinyOS application (called Aislemonitor) is to track the movement of people across an aisle. We consider a sensor network consisting of eight motes shown in Fig. 4. Each mote is equipped with an infrared motion detector. Each sensor generates an electrical pulse based on the difference between the temperature of a heat source and the ambient temperature of the environment. The motion detectors are characterized by their field of view. Based on the sensor measurements, each mote computes an estimate of the positions of the people located in its field of view represented as Gaussian distributions (μ_i, σ_i). The motes are placed so that they have overlapping fields of view and they cover the whole space. Neighboring motes communicate with each other the positions that lie on overlapping fields of view and they eliminate duplicate entries based on the proximity between two distributions. . So if $(\mu_i - \sigma_i <= \mu_j <= \mu_i + \sigma_i)$, the entry (μ_j, σ_j) will be deleted. This tracking algorithm does not use any a priori information (e.g. a dynamical model) for the motion of people walking in the aisle.

Reconfiguration is driven by power constraints on the motes. A Monitor component measures the battery power remaining on each mote, and communicates the power level to the GCM in the base station. If the battery power of the i^{th} mote falls below 5%, then it will send its data (estimated positions) to its neighboring motes for two consecutive time-steps, and it will shutdown. The tracking application in motes i-1 and i+1 will be reconfigured to account for the loss of the mote. These motes start computing the velocity for each person. If the position gets out of their field of view, an estimate is maintained by propagating the Gaussian distribution using the latest value for the velocity.

The simple tracking application described above is used to demonstrate the reconfiguration capabilities of the proposed architecture. The application was modeled in GRATISPlus. The different versions of the application were simulated in TOSSIM [16]. The data required for the simulation was generated using Matlab and provided to each mote through a text file.

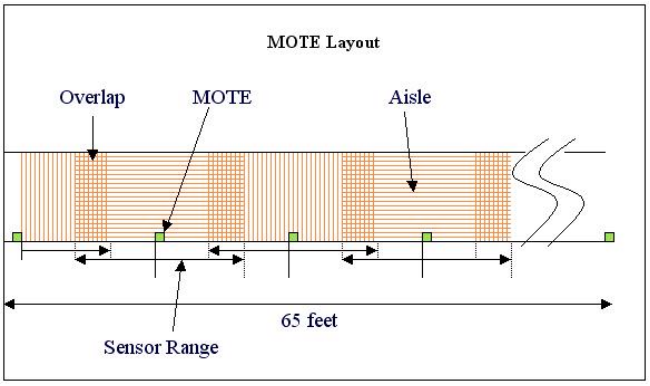

Figure 4. Aisle monitoring application

5.2 Aislemonitor Application Configurations

5.2.1 Aislemonitor #1

Fig. 5 shows the component architecture of Aislemonitor #1 application. The application is driven by a Clock (TimerC) that periodically generates an event. The EventTrigger component monitors these events and performs two tasks that include reading data from a file (sensor) and synchronizing data transfer so that only one mote transfers data at one time. The Data application component contains data structures for storing persistent data about the

people in the aisle and maintaining an accurate list of the people. The SendMessage is the top-level component that takes data from the Data component and transmits it to the destination. The ReceiveMessage component collects the data as it arrives, and passes it onto the Data component for further processing. The directed connections in the figure represent method invocations in each component.

5.2.2 Aislemonitor #2

The functional graph of Aislemonitor #2 is similar to that of Aislemonitor #1. However, Aislemonitor 2 includes additional estimation logic for handling the case where one of the motes shuts down due to loss of power. This additional logic is incorporated in alternate implementation of the 'Data', 'SendMessage', 'ReceiveMessage', and 'EventTrigger' components.

We model these two alternate configurations of the Aislemonitor application in the GRATISPlus modeling environment. Different implementations of the 'Data', 'SendMessage', 'ReceiveMessage' and 'EventTrigger' components are modeled as alternate *modules* within *groups* of the same name. The power constraint is modeled as a *condition* in the top-level Aislemonitor configuration with the following OCL expression: self.power < P, where P is a parameter that is dependent on the current available battery power. The GCM updates the value of P, based on the input from the local monitors and when the available power reduces below a threshold, it triggers a reevaluation of the design space and possible reconfiguration.

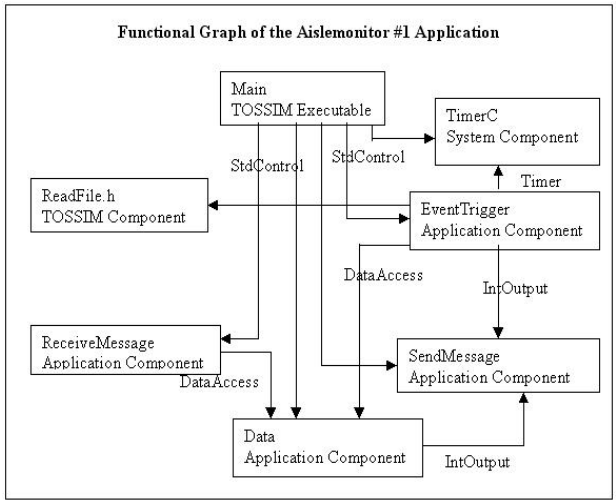

Figure 5. Component architecture of Aislemonitoring

When triggered by the GCM, The GRATISPlus to DESERT mapper converts these models into DESERT's input representation. DESERT then evaluates the power constraint. The power usage of the second configuration is low because it can work with fewer motes, and therefore the power constraint (with a lowered value of P) results in the selection of the second configuration. The output of DESERT is used to create a GRATIS model, where the Groups in the GRATISPlus model are replaced with the selected Module. The GRATIS generator then converts the new model into TinyOS configuration code. Currently, the reconfiguration is executed by manually stopping and restarting the tracking application.

5.3 Performance Evaluation

We 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 only in its field of view are not detected, and (ii) *People double counted*: The tracking algorithm estimates a person in the field of view of the affected mote while he/she has crossed to a neighboring mote.

Mote 2 was shut down in each of the test cases. Fig. 6 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 reduces the "people missed" errors to 10. However, it generated 4 errors due to "people double counted". These errors were generated due to the imperfect nature of the algorithm used to estimate the position of people in the range of the disabled node. Overall, without reconfiguration we had 7.5% errors while with reconfiguration 3.8% errors.

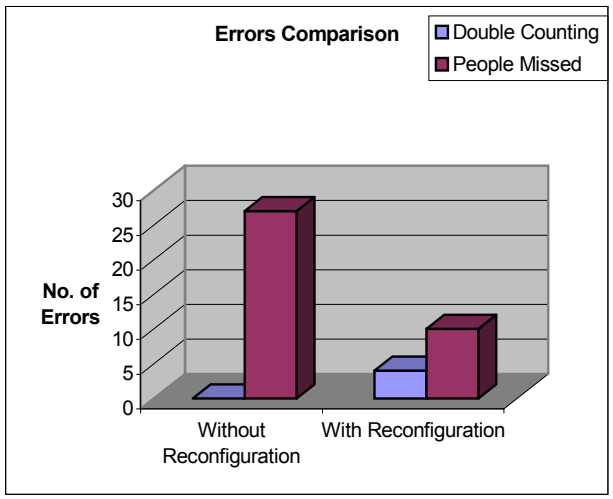

Figure 6. Distribution of Errors in Aislemonitor #1 and Aislemonitor #2

6. DISCUSSION

We presented an approach for constraint-guided software reconfiguration in sensor networks. Our approach requires monitoring the system requirements expressed as formal constraints. These constraints drive the reconfiguration process that takes place in a base station that can communicate to all the sensor nodes. We have demonstrated our approach using simulation results for a simple one-dimensional tracking problem. While the modeling, design-space exploration and reconfiguration tools running on the base station are implemented and tested, implementation of the reconfiguration infrastructure on the motes remains to be done. Although the reconfiguration is achieved manually in our testing implementation, we have demonstrated its advantages. The Reconfigurator component, while relatively sophisticated, does not seem to pose theoretical challenges. However there are additional challenges that need to be addressed. Sensor networks operate in dynamic environments and hence applications must be reconfigured relatively fast. Characterization of worst-case time bounds for the reconfiguration is subject to future research. Robustness of the reconfiguration method is also a significant challenge. In our work, so far, we have assumed static connectivity, which is a very strong requirement. In practice, connectivity will affect the method and especially, the time needed for all the nodes to complete the reconfiguration. The drawback of our approach, the necessity for different switching components for different interfaces, arises because of the static nature of TinyOS. Currently, we investigate these issues by implementing our approach using a Linux-based sensor network that allows dynamic reconfiguration.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge the partial financial support by the Xerox University Affairs Committee, the DARPA IXO MoBIES program, and the DARPA IXO NEST program.

8. REFERENCES

[1] 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.

[2] Blickle T., Teich J., and Thiele L: "System-Level Synthesis Using Evolutionary Algorithms." Design Automation for Embedded Systems, vol 3, pp 23-58, 1998.

[3] Ledeczi A., Davis D., Neema S., Agrawal A.: "Modeling Methodology for Integrated Simulation of Embedded Systems." ACM Transactions on Modeling and Computer Simulation, vol 13(1), pp 82-103, January 2003.

[4] Eames B., Bapty T., Neema S., Abbott B., Chhokra K. : "Model Integrated Design Toolset for Polymorphous Computer-Based Systems," ECBS, pp 72-79, Huntsville AL, April 7, 2003.

[5] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S.: "Model Integrated Tools for the Design of Dynamically Reconfigurable Systems," VLSI Design, vol 10(3) pp 281-306, 2000.

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

[7] Fromherz M. and Conley J., Issues in Reactive Constraint Solving. In: *Workshop on Concurrent Constraint Programming for Time Critical Applications - COTIC 97*, CP'97, Linz, Austria, Nov. 1997.

[8] Laddaga R.: "Active Software," in Robertson P., Shrobe H., Laddaga R. (eds.): Self-Adaptive Software, LNCS 1936, Springer Verlag, February 2001.

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

[10] Mainwaring A., Polastre J., Szewczyk R., Culler D.: "Wireless sensor networks for habitat monitoring", *ACM International Workshop on Wireless Sensor Networks and Applications*, June 2002.

[11] Hill J., Szewczyk R., Woo A., Hollar S., Culler D., Pister K.: "System Architecture Direction for Networked Sensors", *ACM SIGPLAN Notices*, vol. 35(11), Nov 2000, pp 93-104.

[12] Volgyesi P., Ledeczi A.: "Component-Based Development of Networked Embedded Applications," 28th Euromicro Conference, Component-Based Software Engineering Track, Dortmund, Germany, September, 2002.

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

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

[15] http://www.mozart-oz.org/

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

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

[18] Mitchell S., Naguib H., Coulouris G., Kindberg T.: "A QOS Support Framework for Dynamically Reconfigurable Multimedia Applications." 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS '99), Helsinki, Finland, June 1999.