COMPILATION OF DATA-DRIVEN MACROPROGRAMS FOR A CLASS OF NETWORKED SENSING APPLICATIONS

by

Animesh Pathak

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER ENGINEERING)

August 2008

Dedication

To my grandparents, my parents and my teachers.

Acknowledgments

My first feeling of gratitude is for my advisor, Prof. Viktor K. Prasanna, whose guidance over the past several years further enriched this life-changing experience. What I have learned from him will guide my actions throughout my research career. This work would not have been possible without him. I would also like to express my gratitude to my committee members, Prof. Bhaskar Krishnamachari and Prof. Gaurav Sukhatme for their useful insights during this process and for their deep questions that helped in strengthening this dissertation.

Through the course of my Ph.D. I also received excellent guidance and help from a set of wonderful researchers, *viz* the P-Group. Special thanks are due to Sumit Mohanty, Mitali Singh, Zack Baker, Ron Scrofano, Ling Zhuo, Cong Zhang, Amol Bakshi, Yinglong Xia, Hoang Le, and Qunzhi Zhou with whom I had the opportunity to interact closely. I would also like to take this opportunity to express my gratitude for the excellent administrators — Henryk Chrostek, Rosine Sarafian, Aimee Barnard, Estela Lopez, and Janice Thompson — who were always there to support me through this journey. Thanks also to Luca Mottola and Gian-Pietro Picco, collaborating with whom gave my research a fresh perspective.

Apart from research skills, I also learned the art of being a good teacher at USC. I would like to thanks Prof. Gandhi Puvvada, who was my mentor for three semesters, and Diane Demetras, who was always there so solve any teaching-related (and other) issues that I had.

USC also gave me the opportunity to be involved with several excellent student groups, and I would like to thank all the people I came to know through my involvement with the Association of Indian Students, the USC Marathon Team, Pagans and Wiccans at USC, the Natural Path meditation group, the Interfaith council, and the *Vidushak* improv comedy group for giving me a well-rounded personality. Thanks to Reverend Elizabeth Davenport and Arvin Varma for mentoring me in campus affairs.

The friends I made at USC played a large part in supporting me through the highs and lows of the Ph.D. life. I would like to particularly thank Karthik Dantu, Ramakrishna Soma, Jaclyn Kalkhurst, Megan Sim, Vidhya Navalpakkam, Abhijit Chaudhari, Kimish Patel, Joyita Dutta, Revati Kadu, Shobhan Bhatt, Neelakshi Hudda, Maitreyee Tripathi, Rajini Parameswaran, Deepshikha Ganjoo, Gaurav Shrivastava, Sameer Sharma, Samip Bararia, Bhavna Devani, Mary Beth Lutz, and the young lady called PD.

Another source of support in this land far away from home were the several families in Los Angeles who effectively adopted me as their own. Thanks to Vaishali and Chirag Patel, Mamta and Ujj Nath, Gyanam and Anoop Mahajan, Madhavi and Prasad Singamsetty, Vineeta and Manish Pandey and their families for showering me with their love and affection

The people who are perhaps the happiest with this achievement of mine are the ones who matter the most to me — my parents Nilima and Jitendra Nath Pathak, and my brother Pranavesh Pathak. I would not have reached this level were it not for their constant support through the years and their firm belief in me. Gratitude is also due towards my grandparents, uncles, aunts, and the various cousins for their love and support. I am sure their wishes and prayers went a long way in this achievement of mine.

Table of Contents

List of Tables

List of Figures

Abstract

Wireless Sensor Networks (WSNs) are rapidly emerging as a new class of distributed pervasive systems, with applications in a diverse range of domains such as traffic management, building environment management, target tracking, etc. Most, if not all, WSN application design is currently performed at the node-level, with developers manually customizing various protocols to realize their applications. This makes it difficult for the typical domain-expert application developer (e.g., a building system designer) to develop applications for them, and is a hindrance to their wide acceptance. To address this problem, the field of *macroprogramming* has emerged, which aims to provide high-level programming abstractions to assist in application development for WSNs. Although several macroprogramming approaches have been studied, the area of compilation of these macroprograms to node-level code is still largely unexplored.

This thesis addresses the issues involved in the compilation of sensor network macroprograms. The emphasis is on data-driven macroprogramming, where the application is represented as a set of tasks running on the system's nodes - producing, processing and acting on data items or streams to achieve the system's goals. In addition to a modular framework for the overall compilation process, formal models for the task-mapping problem which arises in this context are discussed. Results from optimal result-producing mixed-integer programming techniques and algorithmic heuristics for the above problem are presented. We also present the design and implementation of a graphical toolkit for sensor network macroprogramming.

Chapter 1

Introduction

Since time immemorial, human beings have desired to be able to know more about their surroundings, and to exercise some measure of control over them. The industrial age has given us a wide range of basic *sense-and-respond* systems to partly achieve the above goal, ranging from a simple electric oven controlled by a thermostat to burglar alarms triggered by a tripwire, and traffic lights driven by speed-loops on the streets. The next frontier in this quest to know and control our environment involves larger systems with multiple constituent nodes which act together to perform multiple tasks at the same time. Owing to the large scale of these systems, ease of programming remains a hurdle in their wide acceptance. The work in this dissertation broadly aims to identify and address the issues involved in crossing the above hurdle.

1.1 Networked Sensor Systems

Distributed systems consist of multiple nodes which communicate with each other to achieve systemwide goals. Networked sensor systems are a class of distributed systems where some nodes are equipped with devices which can sense the environment (sensors) as well as affect it (actuators).

The sensors used in these devices can be of various types. Thermal seasons can be and to measure, eg, the temperature in a room or the heat given out by an industrial process. Electromagnetic sensors are used to detect voltages, currents, resistances as well as the presence of large metallic object, e.g. magnetometers can be used to detect the presence of tanks in a battle zone or cars on a highway. Chemical sensors can be used to detect the presence of pollutants in water, air and soil, and mechanical sensors car be used to measure the fow of liquids in pipelines, the strain in the beams of a building or a bridge, and the humidity in a room. Sensors can also be used to measure the light intensity in a room, the ambient noise levels in an area and the speed and location of moving objects, among others. A class of sensors which have emerged as an area of recent interest are *biometric* sensors, which can provide information about vital signs such as blood pressure and heart rate to personal health-monitoring systems.

Actuators are mechanical devices which can be used to control a mechanism or a system. Commonly used actuators include motors (which can drive a wide range of machinery), displays (as seen in buildings or streets), temperature controllers, water sprinklers, pacemakers and drug delivery devices (for medical applications) etc.

Over the past years, advances in VLSI technology have enabled the constituents of these networks to be small, battery-powered, untethered devices which communicate wirelessly with each other, earning them the term *Wireless Sensor Networks* (WSNs). We will use the term WSN to refer to all types of networked sensor systems, although a networked sensor system may contain wired nodes as well as actuators. Since a large number of nodes in modern WSNs are battery operated, and because the terrain they are deployed in may make node-replacement hard (e.g. a forest, oil refinery, nuclear power plant etc.), *energy efficiency* is a prime goal for WSN researchers.

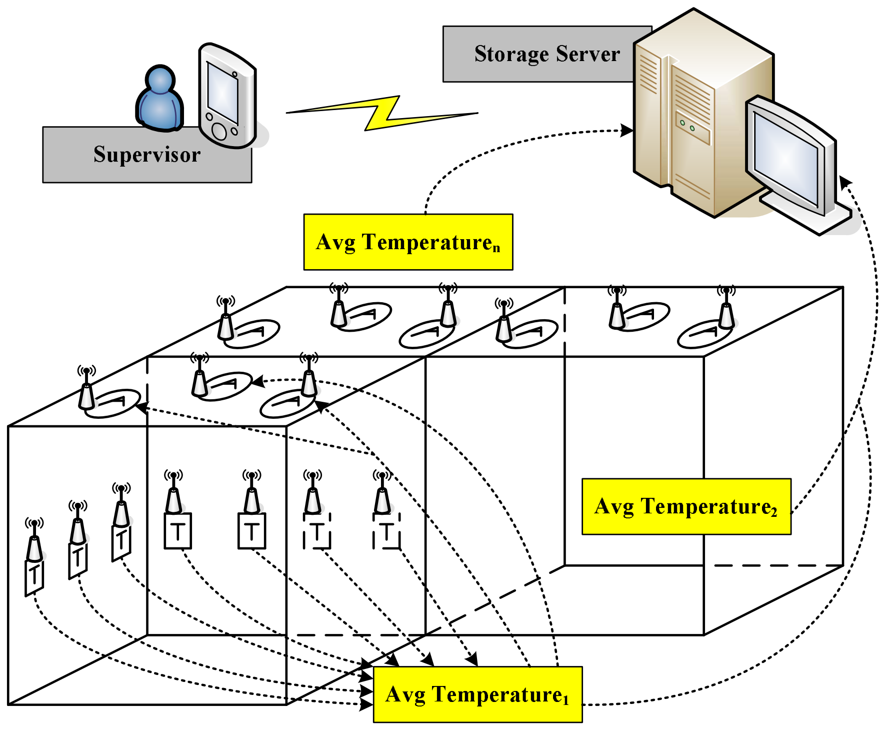


Figure 1.1: Building temperature management using networked sensors

Currently, a wide range of applications are being designed using WSNs, including environmental monitoring [41], target tracking [1, 45], personal health [18, 38], parking garage management [20], building environment management [24], traffic and road management [87, 48], and structural health monitoring [16]. For example, Figure 1.1 shows a scenario where the temperature of the rooms in a building is maintained by interactions between temperature sensors (rectangles marked with a T) and actuators (ovals marked with an $\mathbf{A}$). The average temperature of each room is also sent for storage in a central server which the supervisor can access from time to time.

1.1.1 System Structure

Networked sensor systems of today consist of large numbers of nodes of various types, ranging from the extremely small small TelosB nodes [95] to the more capable Sun

Node	**Processor Speed**	**RAM**	**ROM**
TelosB	8 MHz	10 KB	1 MB
Sun SPOT	180 MHz	512 KB	4 KB
GumStix	600 MHz	128 MB	32 MB
BTNodes	8 MHz	244 KB	132 KB
StarGate	400 MHz	64 MB	32 MB
IMote2	13 - 416 MHz	32 MB	32 MB

Table 1.1: Memory and processing power of common networked sensing nodes

SPOTs [93], GumStix [44], and BTNodes [14], to the even larger micros -servers such as StarGates [92], Imote2 [49] and the ENS Box [67, 32].Table 1.1 lists the capabilities of some commonly used WSN nodes. Depending on the computation and storage capability of the node, the OS support could range from TinyOS [46]], μC/OS-II [68], or Contiki [27] at the lower end to Linux and Windows CE at the higher end. The programmer can also utilize virtual machines like the Maté [59], SUN Squawk virtual machine [91] and VM* [56]. To provide system-level services to these nodes, a host of protocols have been developed by researchers. These include protocols for medium access [51, 73], localization [89], time synchronization [31] as well as routing [55, 83]. For exploring more about the various aspects of networked sensing, the reader can consult several excellent books on the subject [13, 58, 50].

Recently, there has also been a thrust in the field of tiered sensor networks [97]. These systems consist of a hierarchy of WSN nodes of differing capabilities. The lowest tier consists of the small nodes which traditionally run on batteries and are charged with sampling the environment and sending the data over to the great higher level of nodes. The higher tiers of the system are responsible for the tasks of collating the data generated by the sensing nodes, taking decisions based on the data, and the routing necessary for the functioning of the networked sensor system. Some of these nodes at the higher tiers may even be connected to wired power sources and high speed wired networks. A good programming abstraction for such large and complex systems must be able to address the heterogeneity that exists in terms of processing power, sensing and actuation capability, network bandwidth available gas well as the power consumption profile of each constituent node.

1.1.2 Comparison with Traditional Distributed Computing

The idea of multiple constituent nodes of a system collaborating to perform a task is not new, it was first explored in the realm of parallel computing, where several tightly coupled compute nodes work together to solve large computational problems such as weather forecasting, molecular-dynamics simulations, seismology, and remote sensing.

As stated above, parallel computing systems are traditionally tightly-coupled i.e., the member processors are located close to each other (same circuit board, room or building) and there is a dedicated interconnection network for the system. The nodes in a distributed systems, by contrast, have loose coupling between them, *viz.*, the nodes can be located far from each other, and may use public networks to communicate with each other. Examples of large distributed systems include SETI@home [90], Folding@home [35] and the various eScience Grids used for applications such as drug design.

One might be inclined to treat sensor networks simply as wireless distributed systems, acting on nearly the same principles. However, there are several critical aspects in which sensor networks differ from traditional distributed systems:

Spatial Awareness: The nodes of a sensor network are deployed in the environment which they are supposed to monitor and control. e.g., in a building environment management system, each node can be assigned with extra attributes such as the specific room and floor it is in. The applications developed on such a system should both be aware of such annotations and be able to utilize them to easily achieve system specific goals. (e.g. compute the average of the temperatures reported by the nodes in each room, and then compare it with the average temperature in other rooms of the same floor. By contrast, there is no notion of ”my location” in a traditional distributed system.

Nature of Input and Output: Traditional distributed systems work on *transactions* — they take a well defined set of data as input (e.g. a set of numbers), and perform an operation on them to return an output (e.g. the sum of these numbers). Although the processing occurs inside the system, the input and output are both *external* to it. A sensor network, by contrast, is a reactive system. The operations inside the system are triggered by events occurring within the system. e.g. a fire in a room in a building can trigger the activation of sprinklers in it. This reactive processing leads to the need for specialized programming abstractions, which we discuss in detail in Chapter 3.

Optimization Goals: Traditional distributed systems are concerned mostly with the time and space needed to compute the results of their operations on the input data. As a result, modeling the time it takes to compute a basic operation, as well as time taken to communicate the data between the nodes of the system assumes importance in traditional distributed systems. In a sensor network, since a large number of nodes are battery powered the *energy* spent during its operations assumes large importance. Since the nodes of a sensor network communicate using wireless communication, which are more energy consuming than the processing at the nodes, the focus shifts on devising smart strategies so as to minimize the energy spent in the communication of data between the WSN nodes.

1.2 Application Development

The early networked sensor systems were programmed by the scientists who designed their hardware, much like the early computers. However, the intended developer of sensor network applications is not the computer scientist, but the designer of the system using the sensor networks which might be a building or a highway. Throughout this thesis, we will use the term **domain expert** to mean the class of individuals most likely to use WSNs — people who may have basic programming skills but lack the training required to program distributed systems. Examples of *domain experts* include architects, civil and environmental engineers, traffic system engineers, medical system designers etc. We believe that the wide acceptance of networked sensing is dependent on the ease-of-use experienced by the domain expert in developing applications on them.

The various approaches of application development currently available to the domain expert are discussed next.

1.2.1 State of the Art: Node-level Programming

Since their early days, WSNs have been viewed as a special class of distributed systems, and have been approached as such from an application development perspective as well. Consequently, application developers have thus far specified their applications at the level of the individual node where the they use a language such as nesC, galsC or Java to write the program, directly interacting with the node-level services stated earlier, or a middleware [99, 23, 69] that aids in the programming process. The developer can read the values from local sensing interfaces, maintain application level state in the local memory, send messages to other nodes addressed by node ID or location, and process incoming messages from other nodes. However, In all these approaches.

the application developer is responsible for ensuring that these individual finite state-machines executing on the individual nodes of the WSN will interact to produce the desired result.

Owing to the large size and heterogeneity of the systems involved, as well as the limited distributed programming expertize of the domain experts, the above paradigm of *node-level programming* is not easy to use for sensor networks. We believe that this is a large obstacle holding back the wide-acceptance of WSNs. For example, to develop an environment management application in nesC, a commonly used language in WSNs, the developer has to specify the functions at each node in terms of the respective components - one each for sensing the environment, communicating with other nodes, as well as controlling the actuators attached to each nodes. In addition to the above, the application developer is also responsible for ensuring that the distributed application that results from these communicating node-level programs performs the necessary functions as desired, and is also efficient in terms of the energy spent during its operations.

1.2.2 Macroprogramming

Sensor network *macroprogramming* aims to aid the wide adoption of networked sensing by providing the domain expert the ability to specify their applications at a high level of abstraction. In macroprogramming, abstractions are provided to specify the high-level collaborative behavior at the *system level*, while intentionally hiding most of the low-level details concerning state maintenance or message passing from the programmer. As a result of this, macroprogramming is emerging as a viable technique for developing complex embedded applications, as demonstrated by the several

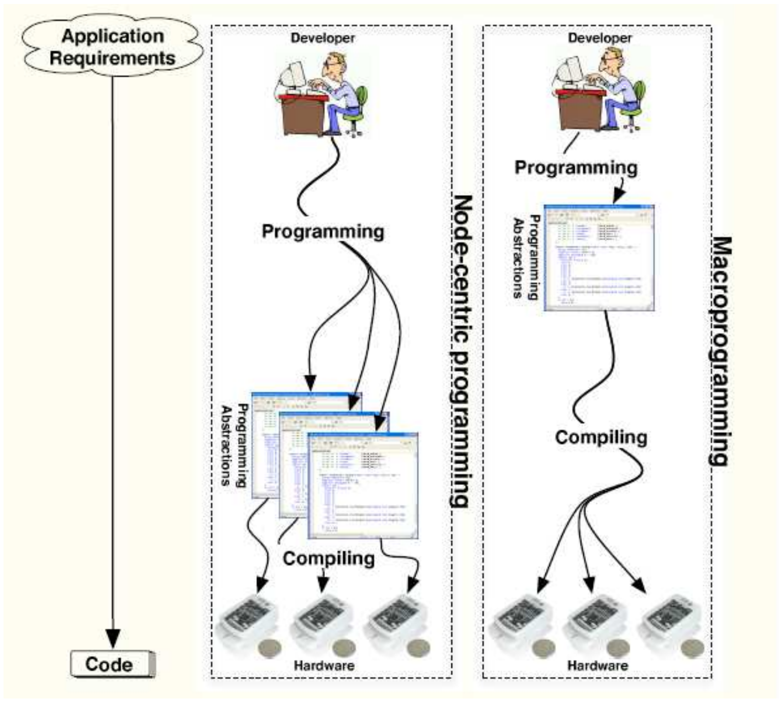


Figure 1.2: Comparing node-centric and macro- programming.

efforts [6, 43, 76] currently underway in this field. Our work is focused on the compilation issues encountered in *data-driven macroprogramming* (discussed in detail in Chapter 3), which allows the developer to specify the functionality of their application in terms of *tasks* that interact with each other only using the *data items* that they produce and consume.

In the context of macroprogramming for WSNs, we define **compilation** as the *semantics-preserving transformation of a high level application specification into a distributed software system collaboratively hosted by the individual nodes*. In [81], we summarized the challenges faced by the designers of compilation frameworks for macroprogramming languages. The process of semantics-preserving transformation itself involves addressing challenges of correct and efficient conversion of representation. In addition, developers should be given the ability to express performance goals for the deployed system (e.g., in terms of expected network lifetime or latency) that the compiler should consider in optimizing the configuration of individual nodes and the allocation of different functionality to them.

As illustrated in Figure 1.2, the ease of design provided by macroprogramming comes at a cost when compared to traditional node-centric programming. In the former approach, application developers reason at a high level of abstraction, while the process of converting the high level representation to that of the individual nodes is delegated to a *compiler*. The higher the level of abstraction, the more work needs to be done by the compiler. This makes the process of generating the final running code significantly different from one solved by the node-level compilers currently seen in WSNs.

1.3 Thesis Contributions and Outline

The aim of this work is to address the issues arising while compiling sensor network macroprograms due to the difference in abstractions of the data-driven macroprogramming language, and the one provided by the runtime systems at the individual nodes. Chapter 2 discusses related work in supporting high-level application development for sensor networks. Chapter 3 establishes the necessary background about data-driven macroprogramming, including details of the Abstract Task Graph (ATaG) macroprogramming language, and the Data-driven ATaG Runtime system (DART), which serve as the input and the output of the compilation process in our work.

Overall, we make the following **contributions** in this thesis:

1.3.1 Compilation Framework for Data-driven Macroprogramming

In Chapter 4 we propose a general framework for compilation which can be used for data-driven macroprogramming languages like ATaG. Our framework breaks down the process of converting the high-level specification to node-level functionality into a set of independent procedures—such as optimizing the placement of functionality on the real nodes, or predicting communication costs. These different stages are connected through well-defined interfaces, that allow for plugging in different modules implementing the various steps of compilation.

We demonstrate the flexibility and generality of our framework by describing an end-to-end solution for compiling ATaG macroprograms. Our proof-of-concept compiler, obtained by instantiating the different modules in our framework, provides the code to be deployed on each node, as well as an estimate of the message passing costs of the same. Moreover, the resulting code can be deployed on real world nodes as well as in a simulation environment.

To evaluate our work, we show results from our work in developing two realistic applications – building environment management (HVAC) [24] and highway traffic management [48], discussed more in detail in Section 3.3. The functionality of our compiler is assessed by inspecting and comparing the auto-generated code against a manually developed version of the same.

1.3.2 Optimization for Task Mapping

The representation of the WSN application as a set of communicating tasks in an important intermediate stage in the compilation of data-driven macroprograms. The mapping of these tasks onto the nodes of the underlying system (details of which are known at compile time) is an important part of the compilation of the macroprogram, and optimizations can be performed at this stage to make the resulting WSN more efficient. In Chapter 5, we show how task mapping in this context differs from the traditional task-mapping problems seen in parallel and distributed computing, in a large part due to the presence of placement constraints for tasks, as well as different (energy-related) optimization goals.

We then provide a framework to model the problem of task-mapping for data-driven sensor network applications, with tasks subject to placement constraints and channels annotated with data-rates. We propose a mixed integer programming (MIP) formulation to obtain task mappings in order to optimize for the energy balance and total-energy minimization goals in the cases where a single route is available between each pair of nodes. Since the formulation is non-linear, we provide substitution-based techniques to linearize the MIPs.

Although the MIP formulations give optimal results, they may take inordinately large times to terminate for large real-world scenarios. To address this issue, we provide greedy heuristics for the two problem instances, along with their worst-case performance analysis.

Finally, we provide formulations of the above problem instances when there are multiple routes available between each pair of nodes in the system. We provide (linearized) MIP-based and greedy techniques to solve these generalized problems.

Our experimental results show the performance comparison between the techniques, using realistic applications and deployment scenarios. Our greedy heuristics are shown to obtain the optimal solution for most of these scenarios, while gaining significant speedups over the MIP technique.

1.3.3 End-to-end Toolkit for WSN Application Development

The overall goal of this thesis is to contribute towards the adoption of networked sensing by the domain expert. Therefore, to provide the domain expert the ease-of-use of the compiler, as well as the efficiency provided by the optimizations, we incorporated our research in a graphical toolkit for WSN application development using data-driven macroprogramming. Chapter 6 discusses the details of various components of our toolkit *Srijan* (named after the Sanskrit word for creation). Domain experts can use our toolkit to graphically specify their networked sensing applications. *Srijan* also allows them to upload a description of the target network to visualize and edit it. Finally, the toolkit is used to compile the macroprogram to node-level codes, and then deploy it on the nodes of the target system. Our experiments show that using *Srijan*, application developers can specify and deploy their applications in a timely fashion, while having to write $\sim 2\%$ of total system code (or $< 10\%$ of application-specific code).

Chapter 7 concludes with a discussion of future directions of work.

Chapter 2

Related Work

2.1 WSN Application Specification and Compilation

Initial programmming of WSNs was done by the nesC [37] language on the tinyOS operating system [46], and other languages such as μC/OS-II [68], and helped a wide research community build and test applications and system components for networked sensing [58, 85, 41, 55]. Over time, tools such as SNACK [42] were developed to support the programmers of such systems, and sensor nodes supporting more traditional programming languages such as Java have also emerged [93]. However, the compilers of all these languages are essentially *node-level* compilers, not very different from the common C compiler used on larger machines.

Researchers have recently worked on addressing the concerns of the end-user of such a system, in efforts ranging from enhancing interoperability between sensor and UMTS networks [88] to macroprogramming approaches for specifying the WSN application itself [43, 76, 6]. Our work focuses on ATaG [6, 8], a data-driven macroprogramming paradigm where the application is specified as a set of interacting tasks. We discuss it more in detail in Chapter 3. *Kairos* [43] is an imperative, control driven macro-programming language where the application designer can write a single program in a Python-like language with additional keywords to express parallelism. A 'centralized' program describes the activities at all nodes in the system and is translated into node-level binaries by a dedicated compiler. Since the program is written in an imperative form, and whether the action will be performed at a particular node or not is decided by conditions mentioned in the macroprogram itself, the issues faced by the compiler are very different from ours. For example, there is no channel composition to be done and no specific tasks to be allocated. The work in [57] extends the above idea to the *Pleiades* language, whose compiler converts the input specification to a control flow graph (CFG), and computes node-cuts representing a set of related actions. The node-cuts are then placed on different nodes, with the runtime system taking care of transferring the flow of control between nodes as needed.

Regiment [76] is a functional programming language, with support for region-based functions like filtering, aggregation and function-mapping. The Regiment primitives operate on a model of the sensor network as a set of continuous data streams. In [75], the authors introduced the TML intermediate language to represent the actions being performed at individual nodes. The authors state that Regiment programs can be seen as *data flow graphs*, with primitives such as **afold** combining functions and data on actual nodes to produce data. Although the functional programming approach of Regiment is very different from the data-driven approach of ATaG, the above similarity (ATaG tasks combine data produced at other nodes to produce more data) might lead to some re-use of our ideas in the compilation of Regiment macroprograms. The work in [77] extends this to the *WaveScript* language which addresses applications working on live data streams. The operator placement step of their compiler is similar to our task mapping stage.

EnviroSuite [63] is an object-based programming system that introduces the *environmentally immersive paradigm*. Its abstractions revolve directly around elements of the environment as opposed to sensor network constructs, such as regions, neighborhoods, or sensor groups. Object instances float across the network following (geographically) the elements they represent. The EnviroSuite Compiler (EIPLC) is a translator that takes EnviroSuite code as input and outputs desired environmental monitoring applications in nesC, which then can be compiled by a standard nesC compiler and uploaded to the motes. In [52], the authors present a programming model that enables opportunistic application flows in pervasive environments. The *clients* in their programming model are very similar to the instantiated tasks encountered during ATaG compilation. If one were to create a higher level of abstraction that would automatically create clients for a given network, the issues faced will be similar to the ones addressed in our work. Also related to our work is the MagnetOS [60] system, which aims at statically partitioning a monolithic program into components that are then distributed over the sensor network, and then migrating the components at run-time for energy efficiency. Although the ATaG programming model is different from it, we indend to keep this system in mind as we add dynamic migration of tasks to the ATaG runtime system.

2.2 Task-Mapping on Networked Systems

A large body of work exists on the problem of mapping tasks of an application onto the nodes of a target system both in the parallel and distributed computing as well as the wireless sensor networking domain. In this section, we present some closely related work from various domains.

Parallel and Distributed Computing: The task mapping problem [11] is a well studied problem is parallel and distributed computing. In [30], the authors have covered a wide range of mapping problems in such systems and approached to solve them. However, they are mostly concerned with optimizing for *latency*, i.e., minimizing the computation and communication time. In addition, the tasks do not have placement constraints. In [64], the authors include placement constraints in their problem statement using a *task preference matrix*. However, they assume that communication costs are paid only by the end-points, and their optimization goal is the total cost that the system endures for the application.

Heterogeneous Systems: In [86], the authors present a genetic algorithm for placing tasks onto a parallel processor. They also provide an extension for the case where not all tasks can be run on all nodes, by way of assigning each node to a class, and associating a class number with each task. Their algorithm is designed to work for a range of metrics, and they focus on the *minimize total execution time* metric in the paper. However, unlike our work, they assume full control over the message routing. In [54], the authors present algorithms based on the best-first A* technique from artificial intelligence for optimal task placement on heterogeneous systems. The placement constraint is specified as a *placement cost metric* for mapping a task to a particular node. Subject to these costs, the nodes are assumed to be capable of executing any task in the application. However, unlike our work, their optimization goal is to minimize the turnaround time. Also, they assume a dedicated interconnection network, and there are no routing overheads for intermediate nodes. Similarly, recent work such as that in [94] focus on scheduling jobs on grids by a Multi-Resource Scheduling (MRS) algorithm using virtual maps and resource potentials. However, they also assume a completely connected network, and no routing costs.

Wireless Sensor Networks: A wide variety of work exists in sensor networks to maximize lifetime by reducing the energy spent, mostly using distributed algorithms for sleep-wake scheduling [84]. The work in [29] achieves energy-balance during data-propagation by deciding in each step whether to propagate data one-hop towards the final destination (the sink), or to send data directly to the sink. This randomized choice ensures that the average per sensor energy dissipation is the nearly the same for all sensors in the network. Task placement on sensor networks has also been addressed recently. One of the early works on this topic is [102], where the authors propose an energy-balanced task allocation for collaborative processing in WSNs. However, unlike our work, they focus on single-hop networks only. Further, our system model is more general than theirs in some respects, since they only consider the case where two tasks cannot share the same node. In [96], the authors have provided task placement approaches for unconstrained task graphs with optimization goals such as minimizing total energy. In addition, they also provide the routes taken by messages. Finally, efforts such as [61] approach the task-mapping problem for WSNs from a protocol-centric point of view, whereas we take a high-level perspective of the problem to determine a good initial task mapping.

WSN Macroprogramming: With the advent of macroprogramming, several approaches have addressed this problem as it arises due to the high-level of applications divided into tasks. [2] proposes a greedy solution to the *service placement problem*, which is applicable to our context of compiling macroprograms. Similar to our case, their application also has task placement constraints, where certain tasks can be placed only on certain nodes. However, they focus only on task graphs that are *trees*, and not general graphs. Further, their algorithm's goal is to minimize the *total* energy of the system, and does not guarantee that a single node will not be over-penalized. The work in [36] solves the generic role assignment problem, where task placements are specified using *roles*. Their algorithm allows ILP solutions of role assignment onto the nodes of the target system, based on a global optimization criteria represented in terms of the number of nodes with a particular role. Unlike their case, our heuristics are meant for solving an offline version of the problem, and the optimization goals more tied to the energy-consumption at the nodes.

2.3 Toolkits for WSN Application Development

On the end-to-end application development toolkit front, despite many years of research in the area, application development for WSNs is still largely done by writing *text-based code* for *individual nodes*, be it nesC on the Mica motes, C on the BTNodes, or Java on the Sun SPOTs. One of the earliest toolkits proposed to reduce the programming effort was the Sensor Network Application Construction Kit (SNACK) [42], which provides a component composition environment that allows developers to define explicit configurable parameters for application-level components. The SNACK user develops applications at the *node-level* using a text-based description of *wiring* between components, several of which are libraries provided by the authors. These programs are analyzed by the compiler to generate maximally-shared nesC expansions, which then have to be deployed just like normal nesC applications. The Flask language [65] facilitates *node-level* programming using data-flow graphs and provide facilities for composing atomic subgraphs across the network using a *flow* communication model. The application is specified in a variant of OCaml, and the behavior of individual processing elements is specified in nesC. The Flask compiler then generates node-level nesC code from the datagraph. This approach of mixed imperative-declarative programming is similar to ATaG, but Flask currently allows application description only at the node-level. On a different direction of research, the

Deployment Support Network WSN development toolkit [28] aims to aid the programmer by collecting data about the deployed sensor network in an over-the-air fashion. In addition to the above, some graphical toolkits have also been proposed for WSN application development. The authors of Viptos [19] allow developers to model and simulate TinyOS applications in a graphical manner. A similar functionality is provided by GRATIS [40] where developers can use GME for easy modeling of TinyOS applications. However, in both these tools, the developer still has to reason at the *node level*, while *Srijan* is geared for enabling developers to think at a much higher level of abstraction.

At a *system-level* of abstraction, one stream of research has focused on treating the WSN as a database, and making it easy for developers to write and deploy data-querying applications on sensor networks. The Task [15] toolkit makes designing and deploying TinyDB query-based application easy, where users can query the sensor data using SQL-like queries, and also provides a visualizer for monitoring the network health and sensor readings. Semantic streams [100] presents each user with a 3D rendering of the sensors in the testbed as well as all predicates that are queryable. The work in [101] builds on it by providing a spreadsheet approach to programming and managing data-querying applications in WSNs. In semantic middleware [12], applications are represented in a graphical interface as composable data sources and inference units which can be connected to retrieve required data by composition engines. jWebDust [17] provides a multi-tier application environment, where different sensor networks can be visualized as one to query the sensed data in a user-friendly manner. While these toolkits help WSN developers by allowing them to reason at a high-level, the developer can specify the application as a query-based system only.

Another domain-specific *system-level* project is EnviroSuite [63], which is targeted at tracking applications. In EnviroSuite, an application contains a list of *objects*, specified in a textual manner, which are abstractions of environment elements. Their toolkit provides keywords and method libraries to define objects and hide the fact that the execution of object methods may need distributed computing across network from programmers. The target code is in nesC, and is deployed in the usual fashion. In the field of environment monitoring, [5] presents a user friendly toolkit, where application developers can specify their application in an Eclipse-based GUI, as well as properties of the target network. The compiler-generated code must then be deployed manually to the target nodes in the system. In contrast, *Srijan* is a more general-purpose tool, which can be used to design and deploy a variety of data-driven WSN applications.

Perhaps the closest tool to our work is VisualRDK [98], which enables the developers of pervasive applications to easily develop applications for heterogeneous systems using a graphical toolkit, where individual tasks can communicate using simple triggers. *Srijan* provides a similar, easy-to-use graphical interface specifically geared towards developing complex applications for sensor networks using data-driven macro-programming.

Chapter 3

Data-driven Macroprogramming

Macroprogramming of WSNs is an active area of research, with several programming paradigms currently being investigated [6, 43, 76], allowing the developer to use a variety of paradigms such as imperative and functional programming to specify their applications. In our work, we focus on the *data-driven macroprogramming* paradigm, where the developers breaks up the functionality of their application into of *tasks* that interact with each other only using the *data items* that they produce and consume, and do not share any state otherwise. We believe that this technique is especially useful in specifying a wide range of sense-and-respond applications [79].

3.1 ATaG: Abstract Task Graph

The specific data-driven macroprogramming technique that we focus on in this work is called the *Abstract Task Graph* (ATaG) [6, 8]. ATaG includes an extensible, high-level *programming model* to specify the application behavior, and a corresponding node-level *run-time support*, the data-driven ATaG runtime (DART) [7]. The compilation of

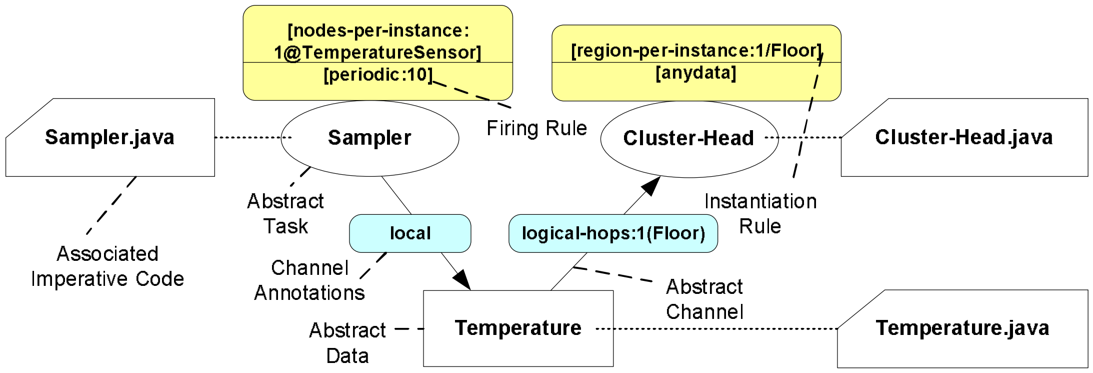


Figure 3.1: ATaG program for data-gathering

ATaG programs consists of mapping the high-level ATaG abstractions to the functionality provided by DART. We now provide some background on these topics, as they represent the inputs and outputs of the transformation process, respectively.

ATaG provides a data driven programming model and a mixed *imperative-declarative* program specification. A *data driven* model provides natural abstractions for specifying reactive behaviors, while *declarative specifications* are used to express the placement of processing locations and the patterns of interactions.

The declarative portion of an ATaG program – a task graph – consists of the following components (see Figure 3.1 for details).

- **Abstract Data Items:** The main currency of information in an ATaG program. They represent the information in its various stages of processing inside a WSN.

- **Abstract Tasks:** These represent the processing performed on the abstract data items in the system. Tasks do not share state with other tasks, and can communicate only by producing and consuming data items. Tasks are annotated with *instantiation rules*, specifying where they can be located, as well as *firing rules*, specifying whether a task is triggered periodically or due to the production of certain data item(s).

- **Abstract Channels:** These connect tasks to the data items consumed or produced by them, and are annotated with logical scopes [72], which express the interest of a task in a data item. In an ATaG program, a data item can only be produced by one abstract task, but can be consumed by many.

For each ATaG task, the developer also specifies the actions taken by the task using imperative code such as C or Java. Note that this code is concerned mostly with the processing of the data that the task has received, and generating the data items that the task will produce. To interact with the underlying runtime system, each task must implement a **`handleDataItemReceived()`** method for each type of data item that it is supposed to process. The task can output its data by calling the **`putData()`** method implemented by the underlying runtime system. Additionally, the developer needs to specify the details of each data item using imperative code.

Figure 3.1 illustrates an example ATaG program specifying a data gathering application [21] for building environment monitoring. Sensors within a cluster take periodic temperature readings, which are then collected by the corresponding cluster-head. The *Sampler* task represents the sensing in this application, while the *Cluster-Head* task takes care of the collection. The *Temperature* data item is connected to both tasks using abstract channels. The *Sampler* is triggered every 10 seconds according to the **`periodic`** firing rule. The **`any-data`** rule requires *Cluster-Head* to run when a data item is ready to be consumed on *any* of its incoming channels. The **`nodes-per-instance:`**q**`@Device`** instantiation rule requires the task to be instantiated once every q nodes equipped with a specific device. According to **`@TemperatureSensor`**, the *Sampler* task in our example will be instantiated on every node equipped with a temperature device. Since the programmer requires a single *Cluster-Head* to be instantiated on every floor in the building, the **`partition-per-instance:1/Floor`** instantiation rule is used for this task. Its semantics is to derive a system partitioning based on the values of the node attribute provided (**`Floor`**). In this case, the programmer requires only *one* task to be instantiated in each partition.

As discussed earlier, the channels in our example program are annotated to express the interest of the producer and consumer tasks. The *Sampler* task generates data items of type *Temperature* kept **`local`** to the node where they have been generated. The *Cluster-Head* collects data not only from its own partition (floor), but also from adjacent ones. The **`logical-hops:1(Floor)`** annotation specifies a number of hops counted in terms of how many system partitions can be crossed, independent of the physical connectivity. Since *Temperature* data items are to be used within *one* partition (floor) from where they generated, they will be delivered to cluster-heads running on the same floor as the task that produced them, as well as adjacent floors.

3.2 DART: Data-driven ATaG Runtime

The node-level code output by the ATaG compiler is designed to run atop a supporting runtime hiding the underlying, platform-specific details. Figure 3.2 depicts the architecture of the Data-driven ATaG Runtime (DART) [7]. The functionality is divided into a set of modules to facilitate customization to various deployments.

The *ATaGManager* stores the declarative portion of the user-specified ATaG program that is relevant to the particular node. This information includes task annotations such as firing rule and I/O dependencies, and the annotations of input and output channels associated with the data items that are produced or consumed by tasks on the node. The *DataPool* is responsible for managing all instances of abstract data items produced or consumed at the node. The *NetworkStack* module is in charge of delivering data across nodes. The routing layer in it provides data-delivery across *logical scopes* [70, 69] by implementing a dedicated routing scheme. In particular, the inputs to this module include the data items and the *scope specifications* those are addressed to. A scope identifies, in a logical manner, the nodes an item is addressed to by referring to the relevant node attributes. For instance, a scope may specify all the nodes running the *Cluster-Head* tasks deployed on the first *Floor* as intended recipients. Other subsystems of the *NetworkStack* are in charge of communication with other nodes in the network, and managing the physical layer protocols. Note that by itself, ATaG does not deal with fault tolerance. However, the runtime system and compiler developers are free to provide the user with an implementation that takes desired fault-tolerance requirements and support them by techniques such as task migration.

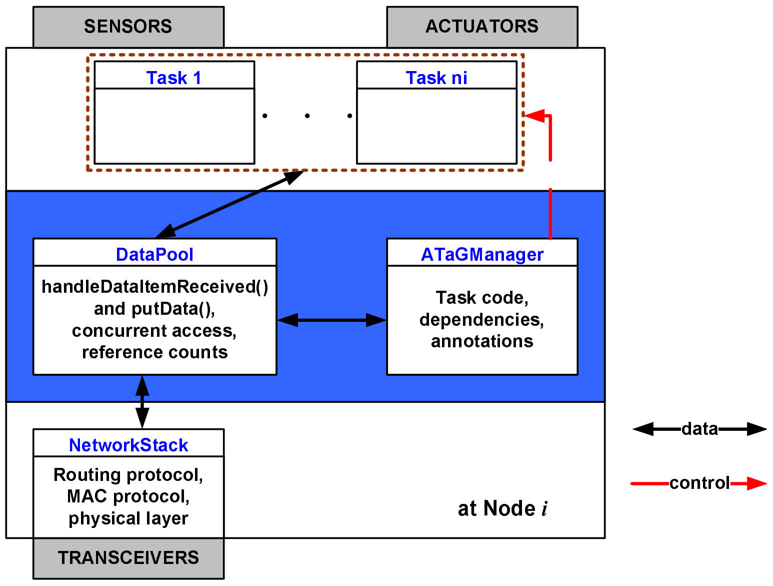

SENSORS
ACTUATORS
Task 1
Task ni
DataPool
handleDataItemReceived() and putData(), concurrent access, reference counts
ATaGManager
Task code, dependencies, annotations
NetworkStack
Routing protocol, MAC protocol, physical layer
TRANSCEIVERS
at Node *i*
data
control

Figure 3.2: DART: Data-driven ATaG run-time system.

3.3 Reference Applications

To evaluate our work on supporting data-driven macroprogramming of sensor network applications, we consider two non-trivial real-world applications.

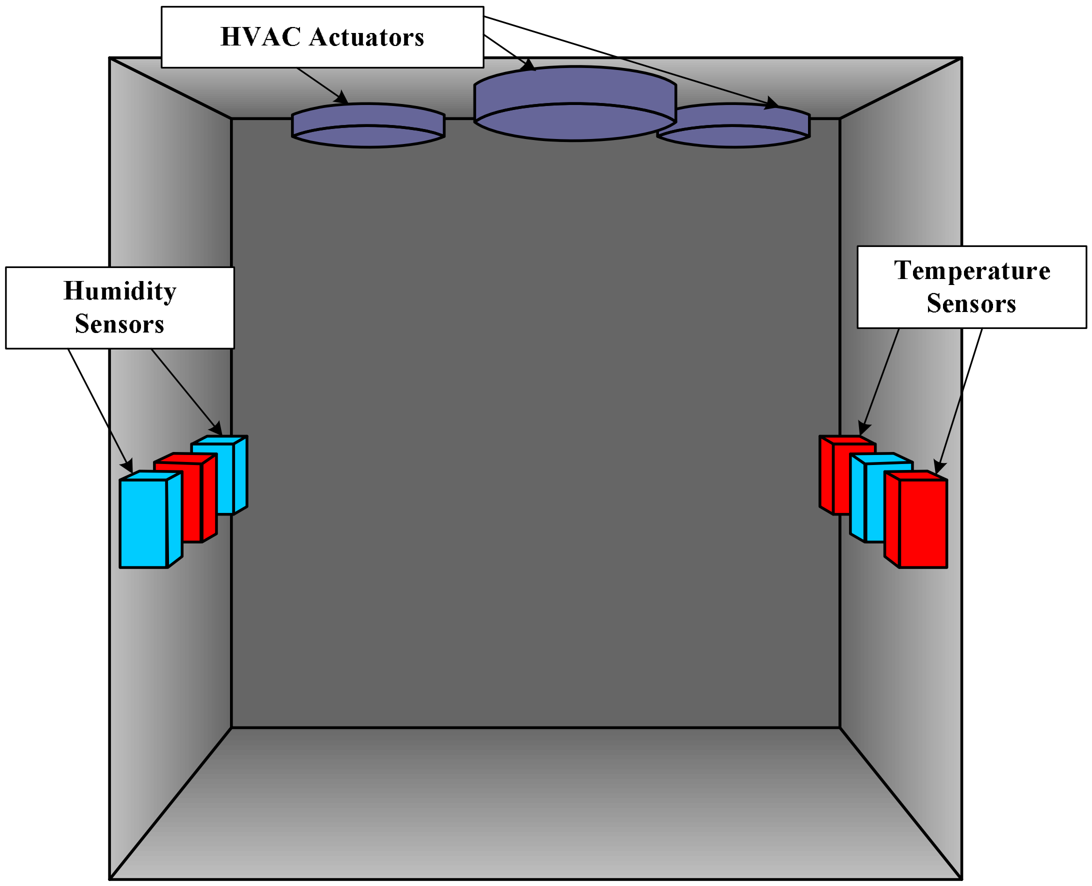


Figure 3.3: Scenario for building environment management (HVAC) application.

3.3.1 Building Environment Management (HVAC)

The first application we discuss is a *building environment management* system [24], whose aim of this application is to regulate the humidity and temperature in each room, based on the current values of those environmental metrics, and achieve a desired steady state. For this purpose, the temperature and humidity readings from all the sensors in each room need to be collected and used to determine necessary actuation. A sample scenario is shown in Figure 3.3 which shows a room in a building equpped with temperature and humidity sensors, as well as HVAC actuators. Note that unlike the application shown earlier in Figure 1.1, this application **a)** does not need to send

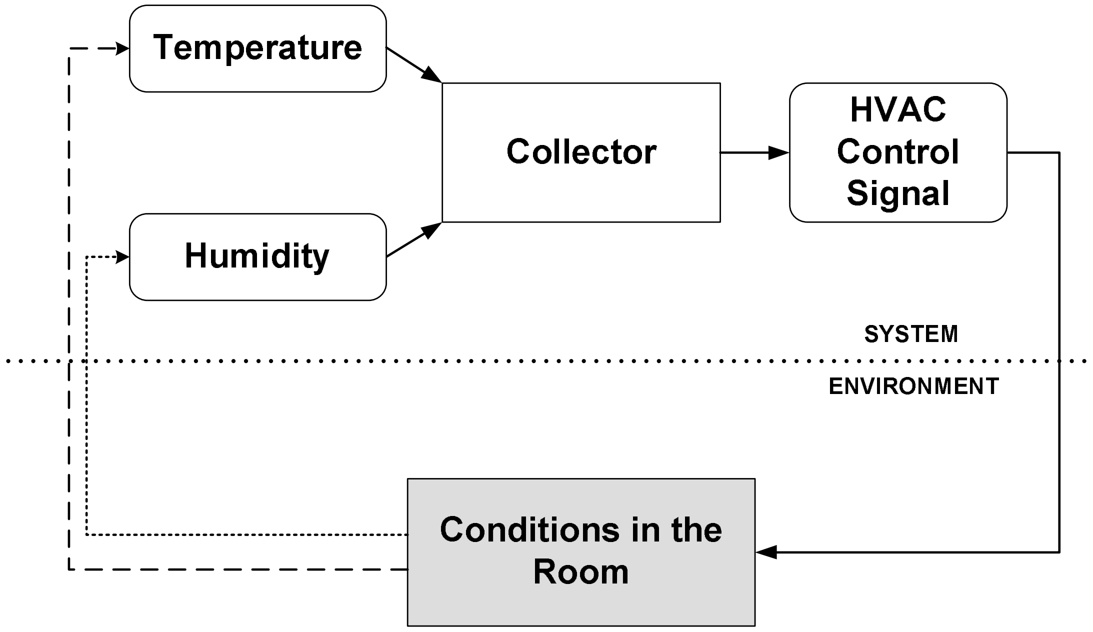


Figure 3.4: Data processing in HVAC application.

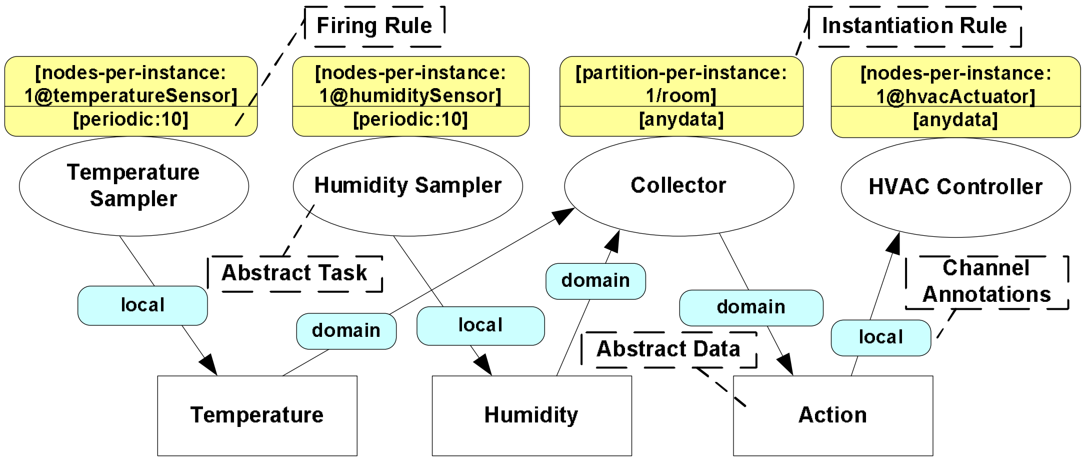

Firing Rule
Instantiation Rule
[nodes-per-instance: 1@temperatureSensor] [periodic:10]
[nodes-per-instance: 1@humiditySensor] [periodic:10]
[partition-per-instance: 1/room] [anydata]
[nodes-per-instance: 1@hvacActuator] [anydata]
Temperature Sampler
Humidity Sampler
Collector
HVAC Controller
Abstract Task
Channel Annotations
local
domain
local
domain
domain
local
Abstract Data
Temperature
Humidity
Action

Figure 3.5: An ATaG program for building environment management.

data to a central server to handle, and **b)** is also responsible for controlling the humidity each room, in addition to ambient temperature.

Figure 3.4 shows the detailes of how the data produced by the sensors is used to determine the needed actuations. For each room, the temperature and humidity values are used to compute the actions needed from the actuators.

Figure 3.5 shows how the HVAC application can be specified using ATaG's data-driven paradigm. The *Temperature Sampler* and *Humidity Sampers* tasks fire periodically and generate the *Temperature* and *Humidity* data items respectively, containing their sensor readings. The **`@TemperatureSensor`** and **`@HumiditySensor`** constructs are used to distinguish nodes with different types of sensing devices, where these tasks are placed. The *Collector* task, instantiated on exactly one node in each room, collects these data items from its *domain*, *viz.* the room it is instantiated in, to compute the desired *Action* for the *HVAC Controller* task, that actually operates the heating, ventilation, and air conditioner (HVAC) devices in the building. As for this, the programmer requires the task to be instantiated on nodes with HVAC devices installed by means of the **`@hvacActuator`** construct.

3.3.2 Traffic Management

The HVAC application discussed above is an example of a real-world use of WSN. However, the communication in it happens only between the sensor and actuator nodes deployed in the same room. ATaG is also capable of describing significantly more complex applications, involving interactions between nodes belonging to different system partitions.

To showcase the complexity of the scenarios that can be addressed using data-driven macroprogramming, we consider a *highway traffic monitoring and control* application, a field where WSNs have gained increasing attention from the research community [48]. Various techniques exist to influence the vehicle movement and improve traffic efficiency both in metropolitan areas and highways. In the latter case, two of the most commonly used solutions are speed signaling [4] and ramp metering [53]. The former aims to control the behavior of traffic by suggesting appropriate speeds,

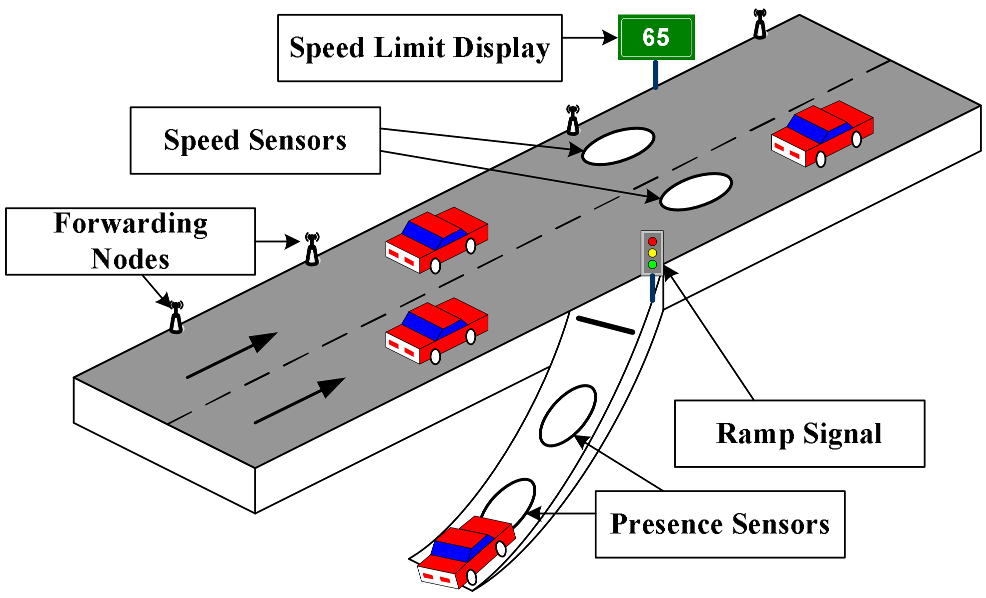


Figure 3.6: Scenario for the traffic management application (one highway sector).

while the latter influences traffic by controlling access to the highway. In these fields, different proposals exist to optimize goals such as pollution and fuel consumption [66].

Scenario: Our reference scenario is depicted in Figure 3.6. Usually, this kind of system is divided into disjoint *sectors* [66], with each sector usually being controlled depending on the current status of the *same* and *neighboring* sectors. In the highway scenario of the figure, a sector is identified by a single ramp leading to the highway, i.e., it spans the portion of highway from a ramp to the following. The system has five main components: i) **speed sensors** installed on the lanes of the highway to measure and report the speeds of vehicles, ii) **presence sensors** installed on the highway ramps to report the presence of vehicles, iii) **speed limit displays** installed one per highway sector to inform the drivers of the recommended speed limit, iv) **ramp signals** installed one per highway ramp to allow or disallow cars onto the highway, and v) **forwarding nodes** to enable wireless communication between the various nodes, installed on the road side at regular intervals.

Figure 3.7 illustrates, from a high-level perspective, the various stages of data processing in the application. Data is first collected from the sensing devices, and a first processing is performed to derive *aggregate measures* – the average speed of vehicles in a highway sector or the average queue length on a ramp. This information is fed as input to an algorithm determining the *actions* to achieve the system objectives, e.g., to maximize the flow of vehicles on the highway. These actions are then communicated to the ramp signals and to the speed limit displays. The specific algorithms employed depend on the goals and metrics of interests. Therefore, a modular approach to the development of this class of applications may be advisable.

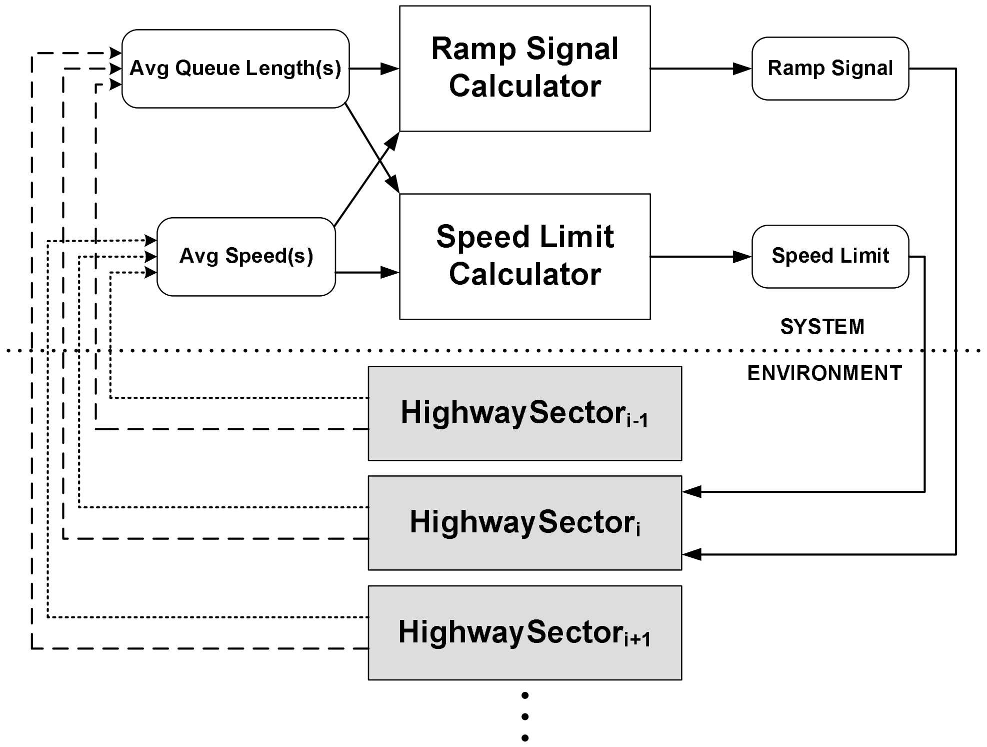


Figure 3.7: Data processing in traffic management.

ATaG Program:

In Figure 3.8 illustrates an ATaG implementation of the traffic management application. All the application information is represented as ATaG data items. The actual

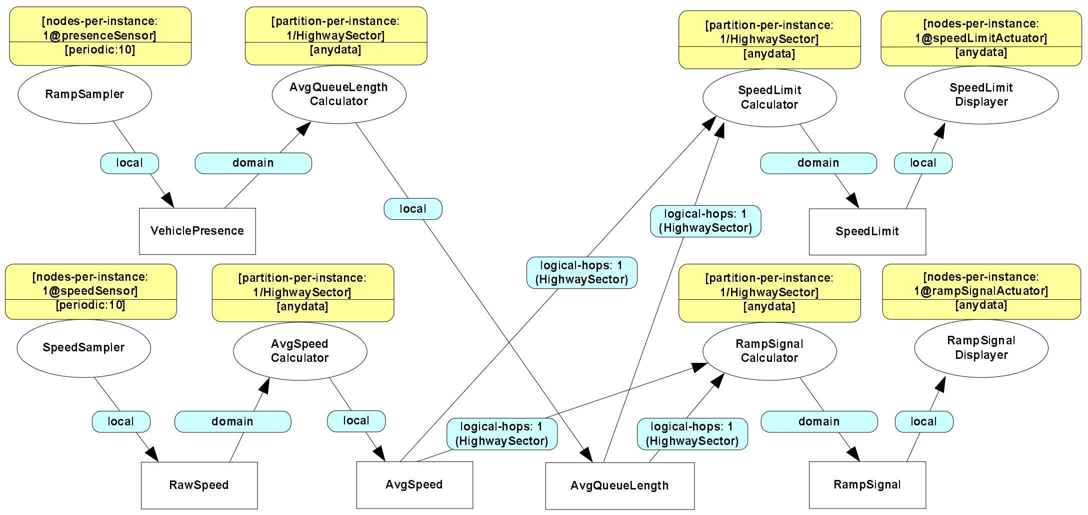

[nodes-per-instance: 1@presenceSensor] [periodic:10]
RampSampler
[partition-per-instance: 1/HighwaySector] [anydata]
AvgQueueLength Calculator
[partition-per-instance: 1/HighwaySector] [anydata]
SpeedLimit Calculator
[nodes-per-instance: 1@speedLimitActuator] [anydata]
SpeedLimit Displayer
local
domain
local
VehiclePresence
logical-hops: 1 (HighwaySector)
SpeedLimit
[nodes-per-instance: 1@speedSensor] [periodic:10]
SpeedSampler
[partition-per-instance: 1/HighwaySector] [anydata]
AvgSpeed Calculator
[partition-per-instance: 1/HighwaySector] [anydata]
RampSignal Calculator
[nodes-per-instance: 1@rampSignalActuator] [anydata]
RampSignal Displayer
RawSpeed
AvgSpeed
AvgQueueLength
RampSignal

Figure 3.8: An ATaG program for highway traffic management.

algorithm determining the actuation part is encapsulated in two tasks: *SpeedLimit-Calculator* and *RampSignalCalculator*, whose inputs are the data produced by tasks deriving the average measures. Once the actuation is determined, it is given as input to the tasks operating displays and ramp signals.

This application showcases the various aspects of application specification that data-driven macroprogamming using ATaG can aid the developer with, namely:

- **Instantiating Multiple Tasks in a Scope:** The *SpeedSampler* task is in charge of gathering the raw data from a speed sensor on a ramp leading to the highway. Therefore, it must run on a node equipped with the corresponding sensing device. To express this requirement, the **`nodes-per-instance:1@speed-Sensor`** construct is used, where **`@speedSensor`** is a placeholder for a membership function $f_{speedSensor}(i) ::= hasSpeedSensor(i)$. Similar constructs are used for *RampSampler*, *SpeedLimitDisplayer*, and *RampSignalDisplayer*.

- **Instantiating a Single Task in a Scope:** The *AvgSpeedCalculator* task takes as input the raw data coming from the speed sensors in a sector, and derives the average speed of vehicles in the same sector. Therefore, we need such a task to be instantiated once per sector. To express this, the **`partition-per-instance:1/HighwaySector`** construct is used. Again, **`HighwaySector`** is a placeholder for a membership function that identifies all the nodes in a specific sector. The compiler generates all possible values of the corresponding node attribute —that describes the sector where a node is placed in the highway— and requires the task to be instantiated on one node in each sector only.

- **Inter-Task Communication:** To bind tasks running in the same **`HighwaySector`**, the **`domain`** annotation can still be used. However, this time it binds to the system partitioning obtained through the **`partition-per-instance`** instantiation rule. Differently from **`area-per-instance`**, this rule determines the different partitions at a logical level, by considering the node attributes instead of the geographical location.

  More generally, the construct **`logical- hops:1(HighwaySector)`** connecting, e.g., the *AvgSpeedCalculator* to both the *SpeedLimitCalculator* and the *RampSignalCalculator* is used to push a data item to a different highway sector. It represents a number of hops counted not on the physical network links, but in terms of how many system partitions (derived from the attribute given in parenthesis) can be crossed. Given the partitioning induced by the **`HighwaySector`** attribute, requiring one logical hop on that attribute means, for an *AvgSpeedCalculator* task, to push a data item to the same, immediately preceding and following highway sectors. Notice how the semantics of specifying zero hops is to not cross any partition, i.e., to push to the same partition where the data item originated. In this sense, the **`domain`** construct actually constitutes a particular case of the more general **`logical-hops`** construct.

3.3.3 Wide applicability of data-driven macroprogramming

The applications just described encapsulates behaviors and interactions seen in a large class of networked embedded applications [80].

These characteristics are grounded in the use of *heterogeneous* nodes, and in the presence of *multiple, concurrent activities* collaborating to achieve the application goals. Developers of these applications must therefore address requirements such as:

- **Multi-stage data processing:** as the raw sensor data is not useful by itself, the system needs to compute the average speed and queue length used to compute the ramp signals and speed limits. This represents a common need in sensor networks when actuation is involved [3].
- **Multiple sub-goals:** to achieve the high-level application objective, e.g., maximize the vehicle throughput on the highway, the system is required to run multiple parallel activities. In our case, regulating the speed of vehicles on the highway and controlling the access to it. This is often required when the system is designed to react to sensed data. For instance, in a different scenario like building monitoring and control, the system is normally required to perform at least three activities [24]: i) indoor environmental monitoring, ii) structural monitoring, and iii) response to extreme events such as fire.
- **Localized interactions:** each of the aforementioned sub-goals usually involves only a specific part of the system. For instance, controlling the speed in a specific highway sector relates to the sensors deployed on the lanes of three neighboring

sectors only. Keeping the processing close to where data is sensed has been long recognized as an effective approach to save energy and achieve more efficient implementations [33, 3].

- **Heterogeneity handling:** various types of nodes are to be employed, with different characteristics and various devices attached. In our scenario, presence and speed sensors are employed along with nodes controlling the speed limit displays and ramp signals. Similarly, in building control and monitoring different kind of sensors are used as well, e.g., temperature, humidity, and smoke sensors [25],

We chose to focus on data-driven macroprogramming using ATaG for our work partly because most of the existing WSN programming frameworks cannot meet the above requirement easily. In first place, they do not provide programming constructs to enable a clear modularization of different activities or consecutive stages of processing. As a result, breaking the high-level application goal into smaller collaborating activities becomes hard to achieve. More importantly, they do not provide support for *heterogeneity*. It is therefore difficult to identify the portion of the system concerned with a specific activity. For instance, developers cannot map a specific processing to the nodes equipped with a given sensing device.

Chapter 4

Compilation Framework for Data-driven Macroprogramming

4.1 Compilation Process

In this section, we provide a formal definition of the process of compiling data-driven macroprograms to node-level code using the data-gathering application given in Figure 3.1 as example.

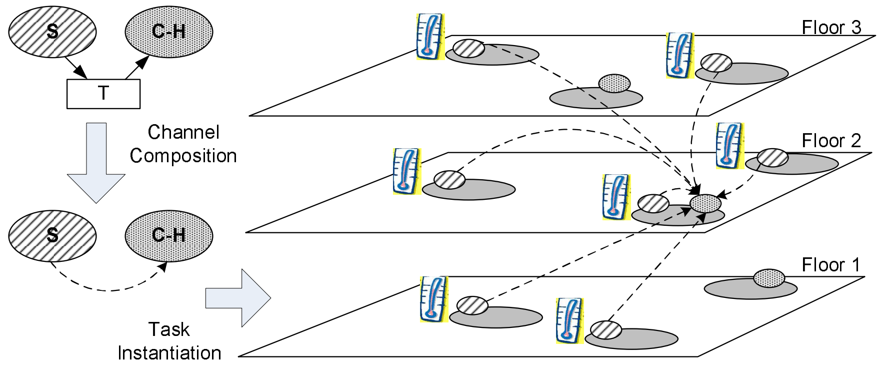


Figure 4.1: An example illustrating the compilation process of our sample program.

4.1.1 Input

The input to the compilation process consists of the following three components.

Abstract Task Graph (Declarative Part): Formally, an abstract task graph $A(AT, AD, AC)$ consists of a set AT of abstract tasks and a set AD of abstract data items. The set of abstract channels AC can be divided into two subsets – the set of *output channels* $AOC \subseteq AT \times AD$ and a set of *input channels* $AIC \subseteq AD \times AT$. In our example, the *Sampler* is AT_1 and *Cluster-Head* is AT_2, while *Temperature* is AD_1. AOC is $\{AT_1 \rightarrow AD_1\}$ and AIC is $\{AD_1 \rightarrow AT_2\}$.

Imperative Code for Each Task: For each task and data item, the developer provides imperative code, which describes the actions taken at the host node when a task fires, and the internal details of the data item.

Network Description: For every node in the target network N, the compiler is also given the following information:

- j: its unique ID.
- S_j: the list of sensors attached to j.
- A_j: the list of actuators attached to j.
- R_j: a set of $(RegionLabel, RegionID)$ attribute-value pairs to denote its membership in the regions of the network (e.g., $\{(\text{Floor}, 5), (\text{Room}, 2)\}$).

Runtime Library Files: These files contain the code for the basic modules of the runtime system that are not changed during compilation, including routing protocols etc.

4.1.2 Output

The goal of the compilation process is to generate a distributed application for the target network description commiserate with what the developer specified in the ATaG program. The output consists of the following parts:

Task Mapping: The compiler must decide on the mapping to allocate the instantiated copies of the abstract tasks in AT to the nodes in N so as to satisfy all placement constraints specified by the developer.

Customized Runtime Modules: The compiler must customize the *DataPool* of each node to contain a list of the data items produced or consumed by the tasks hosted by it. It also needs to configure the *ATaGManager* module with a list of composed channel annotations, so when a data item is produced, the runtime can compute the constraints imposed on the nodes which are hosting the recipient tasks for it.

Cost Estimates: The compiler also provides an estimate of the running cost of the application on the target deployment to provide feedback to the application developer. Note that the actual nature of the cost estimates returned can vary depending on the developer's needs. The costs returned may simply represent a measure of the communication overhead involved, e.g., in terms of messages exchanged per minute on a system-wide scale. Alternatively, finer-grained information may be computed, such as the expected per-node lifetime.

4.1.3 Process Overview

The *abstract* nature of the task graph is precisely what provides the application developer the desired high-level of abstraction needed to easily develop large and complex sense-and-respond applications for networked sensing systems. However, converting this high level specification to a distributed application while preserving program semantics can be quite challenging. In our approach to the data-driven macroprogram compilation problem, we envisage the following major steps.

Composition of Abstract Channels: Owing to ATaG's purely data-driven programming model, the developer only specifies relations between tasks and the data items they are producing (via AOC) and consuming (via AIC). While this provides a clean model to the application developer, traditional task allocation techniques work on task graphs with *direct dependency* links between tasks. To address the problem of generating such task graphs, we convert each *path* $AT_i \rightarrow AD_k \rightarrow AT_j$ in the abstract task graph to an *edge* $AT_i \rightarrow AT_j$.

Instantiating Abstract Tasks: The annotations of the abstract tasks in AT allow the developer to design one macroprogram for a variety of deployments. For example, the developer does not need to worry about the number of floors in the building, because he can use the **`region-per-instance:1/Floor`** instantiation rule. After the channels are composed, the compiler has the responsibility of expand this compact representation of the tasks into a full-fledged task graph that truly represents the data-processing happening in the system.

We introduce the **instantiated task graph** (ITaG) as the internal representation used for this stage of the compilation process. It consists of multiple copies of each abstract task specified in the ATaG program, each ready to be assigned to individual nodes. The (directed) edges of the ITaG connect each task to the tasks that depend on it, i.e., the tasks that **a)** copies of abstract tasks that consume the data item produced by it, and **b)** belong to the logical scope specified by the constraints in the connecting composed channel. Formally, the ITaG $I(IT, IC)$ is a graph whose vertices are in a set IT of instantiated tasks and whose edges are from the set IC of instantiated channels. For each task AT_i in the abstract task graph from which I is instantiated, there are

$f(AT_i, N)$ elements in IT, where f maps the abstract task to the number of times it is instantiated in N. $IC \subseteq IT \times IT$ connects the instantiated version of the tasks. The ITaG I can also be represented as a graph $G(V, E)$, where $V = IT$ and $E = IC$. Additionally, each IT_j in the ITaG has a label indicating the subset of nodes in N it is to be deployed on. This overlay of communicating tasks over the target deployment allows us to use modified versions of classical techniques meant for analyzing task graphs.

For example, for the appliation in Figure 4.1, since there are seven nodes with attached temperature sensors, $f(AT_1, N) = 7$, following the **`1@Temperature-Sensor`** instantiation rule of the *Sampler* task. Similarly, $f(AT_2, N) = 3$, since the *Cluster-Head* task is to be instantiated once on each of the three floors. The figure shows one allocation of the tasks in IT, with arrows representing the instantiated channels in IC (we have showed channels leading to only one instance of AT_2 for clarity). Note that the although the ITaG notation captures the information stored in the abstract task graph (including the instantiation rules of the tasks and the scopes of the connecting channels) it does not capture the *firing rules* associated with each task. The compiler's task involves incorporating the firing rule information while making decisions about allocating the tasks on the nodes.

Task Mapping: This task graph with composed channels is then instantiated on the given target network. Figure 4.1 illustrates an example of a target network. The nodes are on three different floors, and those marked with a thermometer have temperature sensors attached to them. In this stage, the compiler computes the mapping $M : IT \rightarrow N$, while satisfying the placement constraints on the tasks.

Customization of Runtime Modules: Based on the final mapping of tasks to nodes, and the composed channels, the *Datapool* and *ATaGManager* modules are configured for each node to handle the tasks and data items associated with it.

4.1.4 Challenges

The various stages described above each pose their own set of challenges. Since the channels in ATaG have logical scopes associated with them, the process of composing channels results in the (composed abstract channel) CAC_{ijk} being annotated with the union of *three* constraints. The first is that the node should have task AT_j assigned to it. The second(third) constraint is obtained by combining the instantiation rule of $AT_i(AT_j)$ with the annotation on the abstract channel connecting it to AD_k. For instance, in our example, after composition, AC_{121} is $\{$*(Cluster-Head is instantiated)* && (*Floor = Floor of Sampler or* $\pm 1)\}$. Depending on the complexity of scopes used in the channels, the resultant constraint can be further simplified by set operations to get a more compact constraint for the composed channel.

During the creation of the ITaG, maintaining the connections between the instantiated tasks in accordance with the placement rules on the tasks and the scope annotations on the channels is of utmost importance, and can be time consuming if not done efficiently. Finally, an added complexity in the compilation process is brought by the large space of *optimizations* possible in the process to meet the user-specified performance goals (e.g. energy efficiency). Note that although tasks are assigned fixed locations at the end of the compilation process, *task migration* can happen later if the the underlying system supports it. Even in such situations, a *good* initial task placement by a compiler using global knowledge can go a long way in creating efficient systems.

In the following section, we describe how the components of the compilation framework work to produce the outputs from the inputs, using the ITaG notation internally, and the details of our implementation of the ATaG compiler.

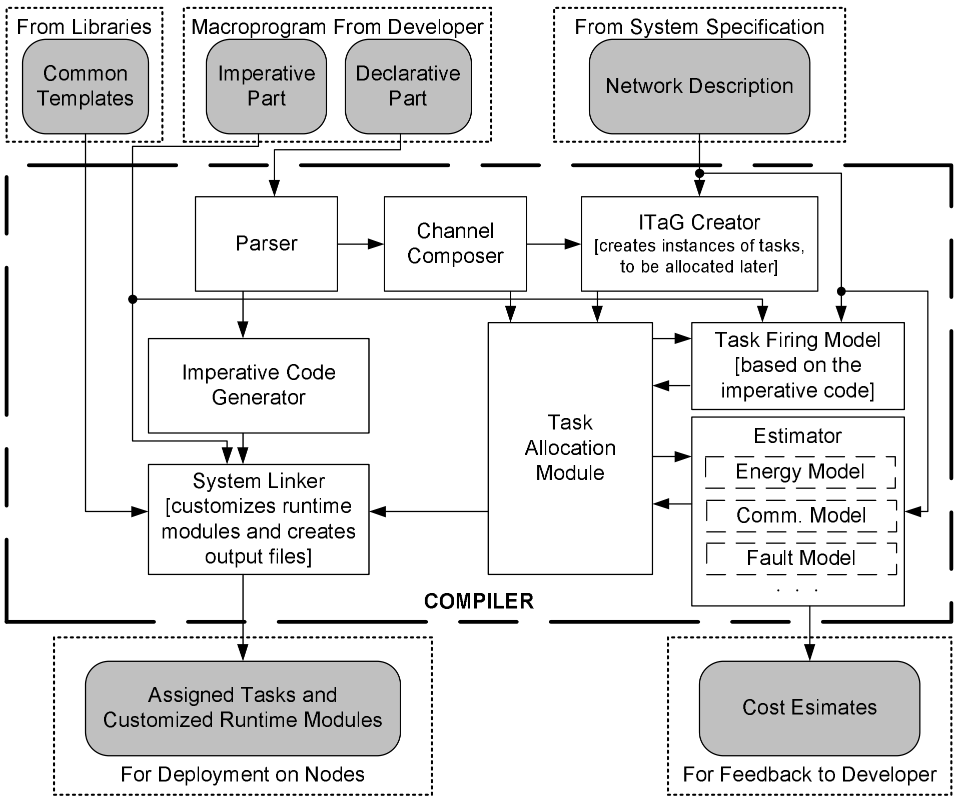

From Libraries
Common Templates
Macroprogram From Developer
Imperative Part
Declarative Part
From System Specification
Network Description
Parser
Channel Composer
ITaG Creator [creates instances of tasks, to be allocated later]
Imperative Code Generator
Task Allocation Module
Task Firing Model [based on the imperative code]
Estimator
Energy Model
Comm. Model
Fault Model
System Linker [customizes runtime modules and creates output files]
COMPILER
Assigned Tasks and Customized Runtime Modules
For Deployment on Nodes
Cost Esimates
For Feedback to Developer

Figure 4.2: The ATaG compilation framework.

4.2 Compilation Framework

ATaG is designed to enable the addition of domain-specific constructs, and customize the abstractions offered depending on the application requirements. This requires a flexible and extensible approach to the compilation problem. Ideally, the system designer should be given the ability to add new language constructs by implementing the required mappings without modifying any of the pre-existing compilation mechanisms. For instance, creating a new instantiation rule should not require modifications to the algorithms used to map tasks to nodes using an existing rule.

To address this issue, we first identified the different steps involved in the compilation of ATaG programs by factoring out orthogonal concerns and mechanisms. Next, considering the decomposition obtained, we designed a modular compilation framework, upon which we based the construction of our ATaG compiler. In this section, we describe the different modules of our framework (illustrated in Figure 4.2), based on the problem definition of Section 4.1. We encapsulated the compilation stages identified by us in separated modules, and defined generic interfaces between them so as to minimize inter-module dependencies. Our current prototype implementation has 2677 lines of non-commented Java code. The modules are as follows:

Parser: The parser converts text files containing the declarative part of the program to an internal representation that is then used by the other modules. This process also involves a syntax check where errors such as duplicate task/data names and the existence of more than one producer task for one data item are identified and reported to the programmer.

In our current implementation, the declarative part of the ATaG program is specified using XML. This will allow an easy integration of tools for the automated generation of XML specifications from graphical representations. Our parser module is a simple XML parser that performs the aforementioned checks, assigns unique IDs to tasks and data items, and populates an internal data structure with the information.

Imperative Code Generator: Based on the parser output, the imperative code generator creates a set of files containing the basic declaration of the variables associated with each task and data items. The imperative part of the code provided by the programmer can then be plugged into these templates.

In our prototype implementation, the imperative part of an ATaG program is expressed using Java. As such, our current code generator creates Java files with unique numerical constants for each abstract task and data item corresponding to their id. Then, it creates a separate class for each abstract task with basic functionality filled in (e.g., a thread instance with a loop for periodic tasks).

Channel Composer: Based on the declarative part of the ATaG program returned by the parser, this module performs the *composition* of channels to and from each data item to form edges of the ITaG, as described in Section 4.1.

Depending on the actual channel annotations supported, our prototype implementation may perform a range of operations, from a simple concatenation to complex operations that also consider the instantiation rules of the producer/consumer tasks.

ITaG Creator: Based on the network description and the output of the channel translator, the ITaG creator first computes the number of distinct *target regions* for each task, i.e., the set of candidate nodes for hosting a given task. For instance, tasks instantiated with **`nodes-per`**
`-instance:x` as instantiation rule have the entire system as target region. For tasks assigned by **`partition-per-instance:x/PLabel`**, each set of nodes with the same value for **`PLabel`** is a target region (e.g., each node in Floor 5). The ITaG creator then instantiates the required number of copies of each abstract task, attaching metadata to each instantiated task signifying its target region. The ITaG creator also computes the edges in this new graph, based on the composed channels. Note that, at this stage, tasks are instantiated but *not yet* assigned to nodes. That is done by the task allocator module, discussed next.

Our implementation of this module performs the above operations using the network description read from a text file containing basic information on the nodes, e.g., their identifier, and set of attributes describing their characteristics, such as sensing devices installed.

Task Allocation Module: As such, the allocation module is one of the most important parts of the compilation process, since it is responsible for computing a mapping from the set of instantiated tasks to the set of nodes. Note the task instantiation rules can be characterized as either *fixed* location (e.g., **`nodes-per-instance:1`**) or *variable* location (e.g., **`nodes-per-instance:3`**), depending on whether or not there is a unique way of instantiating the copies of a task given the network description. In this respect, an extremely large problem space exists depending on the annotations used, metrics to be optimized, and properties of the network. To perform its job, the allocation module relies on two further modules—the estimator and the task firing model–described next.

In our implementation, this module performs task allocation in two passes. In the first pass, it assigns all the tasks with *fixed* locations. In the second pass, it assigns *variable* location tasks. For the latter, one of our initial implementations used a random task-assignment policy, with each node in the target region having an equal probability of hosting the instances of the task. However, due to the generality of our framework, more sophisticated mechanisms can be plugged in to achieve performance goals specified by the application designer. We have also developed several task-mapping algorithms for improved performance [82].

Estimator: Taking as inputs the network description and the task placement returned by the allocation module, the estimator computes the cost metric returned at the end of the compilation process. Our framework gives great flexibility in instantiating this module, as its interface is designed to be generic w.r.t. the nature of information required. This allows application developers to explore the trade-off between the *quality* of the estimate obtained, and the *time* required to obtain it. For instance, during the early design stages it is usually helpful to have a quick estimate of the communication costs, so that many alternative solutions can be explored. In this case, a simple but fast *estimation algorithm* can be employed that does not account for message losses. Conversely, when the application developer is to fine-tune the application, an actual simulation of the deployed application can be run within the estimator.

In our prototype system, we implemented both ends of the spectrum. Specifically, we realized a naive estimator returning communication costs as if all the tasks produced data when fired and the underlying routing mechanisms were able to identify the optimal message routes. On the other hand, we also implemented a wrapper around SWANS/Jist [9]: a simulator able to run unmodified Java code on top of a simulated network. This plug-and-play capability highlights the power of our framework.

Task Firing Model: It would appear that if we know the exact paths taken by the data items, we can precisely estimate the cost of running a given task allocation. However, not all instantiated tasks produce data when they fire. For instance, although a *Temperature Sampler* task may produce a *Temperature* data item whenever it fires, an *Alarm* task may or may not produce an alarm depending on whether or not the temperature of the region is high enough. The task firing model's function is to assign probabilities to the firing of various tasks in the program. Although this module is not mandatory for a working compiler, various approaches can be used to obtain the needed information - ranging from the developer providing profiling data obtained from previous runs of the system, to static code analysis techniques [22, 10].

System Linker: At the end of the whole process, the linker module combines the information generated by the various modules of the compiler into the code to be deployed on the nodes of the target system. More specifically, it configures the *ATaG-Manager* and *DataPool* modules in the node-level run-time depending on the task and data items handled at each node, and merges the imperative code provided by the application developer with the templates generated by the imperative code generator.

In our implementation, the output of this module is a set of Java packages for each node. Note that these files are not binaries. They still need to be *compiled* in the classical sense, but that can be done by any node-level compiler designed for the target platform.

4.3 Evaluating the Compilation Framework

To demonstrate the effectiveness of our prototype compiler, we consider the two real world applications discussed in Section 3.3, and report on the functionality of the code generated, as well as the performance of the compilation process.

Code Functionality: We hand-coded the logic for both applications described in Section 3.3 to perform simulation studies on the underlying routing mechanisms [71]. The hand-written code also allowed us to verify the functionality of our compiler, by comparing the automatically generated code with the one we used in the aforementioned studies. Indeed, by comparing the simulation logs obtained using the SWANS/Jist [9] simulator, we confirmed that the compiler-generated code is functionally equivalent to the hand-written version. The specific code samples can be found at [78].

Settings for Performance Studies: We record the *time* and *memory* taken to compile the above ATaG programs. Since our task firing model assumes that all tasks produce data when fired, the specific imperative code of the tasks does not influence the complexity of compilation. Rather, the compiler's performance is mainly dictated by the declarative part of an ATaG program and the characteristics of the deployment environment. More specifically, we recognized the following factors are pivotal in determining the time/memory taken to compile:

1. the number of abstract *tasks*, *data items*, and *channels*,
2. the nature of *instantiation rules* and *channel interests*, and
3. the *number of nodes* specified in the network description.

The complexity of the compilation task comes from different sources. The effort in composing channels is dependent on the actual channel annotations used, as well as the number of channels themselves. The ITaG creation stage becomes more

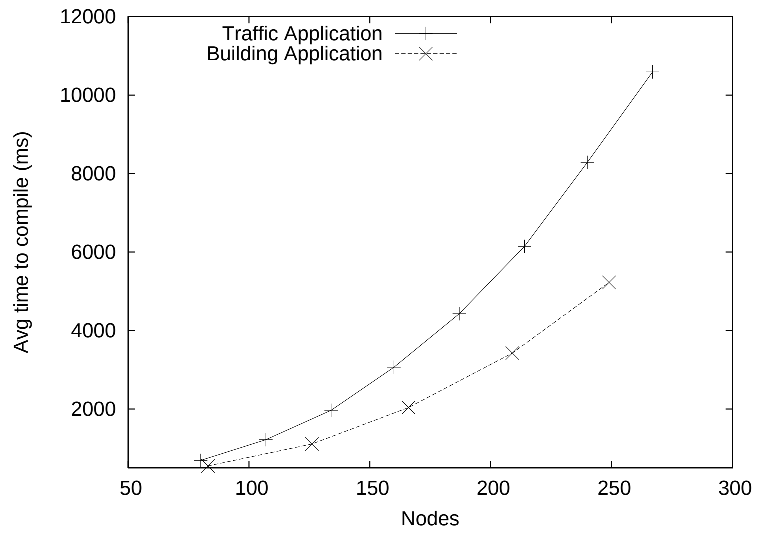
	Building	**Traffic**
Abstract Tasks	4	8
nodes-per-instance:x@PLabel	3	4
partition-per-instance:x/PLabel	1	4
Abstract Data Items	3	6
Abstract Channels	6	14
local	3	6
domain	3	4
logical-hops:1(PLabel)	0	4

Table 4.1: Complexity of the task graphs of sample applications.

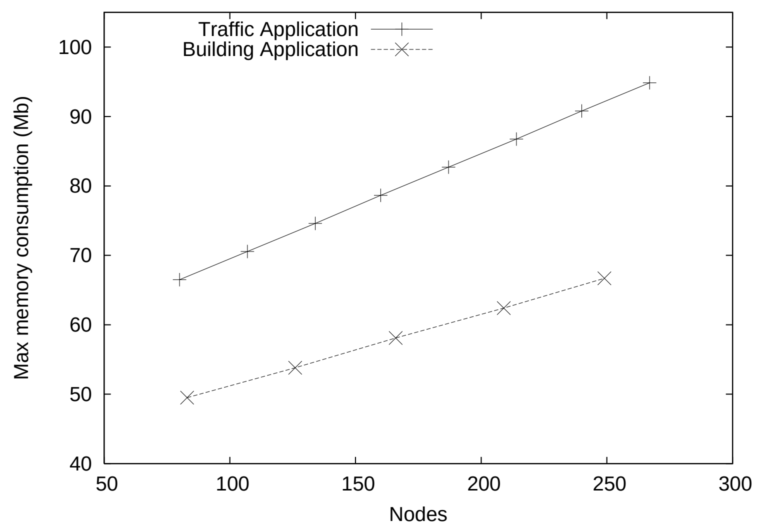
complex as the complexity of the network grows. Note that this includes the number of logical regions the network can be divided into, as well as the variation in the attributes of the nodes. The size of the problem addressed by the task allocation module depends both on the network size as well as the constraints used in the program. For instance, placing a task whose instantiation rule is in the form **`partition-per-instance:x/PLabel`** requires more processing than placing a task with **`nodes-per-instance:1`**. All this in turn affects the performance of the system linker as it customizes the run-time on each node. Figure 4.1 reports the values of these factors seen in our sample applications.

In our tests, the compilation framework has been instantiated with the prototype implementations we described in Sect. 4.2 for each module. In particular, we have chosen to employ the näive estimator and an *always-firing* task firing model. For each test we performed, we repeated the compilation process 500 times to account for fluctuations due to concurrent processes. The experiments were on a Pentium IV HT 3.2 Ghz running Gentoo Linux 2.16.15, using the DJProf [26] profiler.

Performance Results: Figure 4.3 illustrates the performance of our compiler as a function of the number of target nodes. As expected, the time taken to compile an ATaG program grows quadratically as the number of nodes increases. This is due to



(a) Time taken to compile.



(b) Maximum memory consumed during compilation.

Figure 4.3: Compiler performance.

the naive estimator we used, that computes the all-to-all shortest path with an algorithm whose time complexity is quadratic w.r.t. the number of vertices. However, fairly large instances can be compiled in reasonable time. For instance, slightly more than ten seconds are needed to compile the traffic application for a target system with > 250 nodes.

In addition, the memory consumed during the compilation process exhibits a linear increase with respect to the number of nodes in the deployed system. The source of this behavior is in the data structures we employed in the ITaG creator and allocation modules, that allocate a fixed amount of data for each target node. The memory consumed is always well within the limits of standard desktop PCs (< 100 MB).

Chapter 5

Energy-efficient Task-mapping for Macroprogram Compilation

5.1 Introduction

As detailed in the previous chapter, mapping of the tasks of the intantiated task graph onto the nodes of the underlying system (details of which are known at compile time) is an important part of the compilation of the macroprogram, and optimizations can be performed at this stage to make the resulting WSN more efficient.

We note that task mapping in this context differs from the traditional task-mapping problems seen in parallel and distributed computing in several aspects:

1. The task graph here is *constrained* in the sense that some tasks have a one-to-one correspondence with the nodes in a system, while the placement constraint of others may not be as strict. For example, a temperature sampling task can be placed only on nodes with temperature sensors attached to them, while the task that computes the average of temperature readings in a room has much relaxed mapping constraints.

2. Often in the classical task-mapping scenarios, tasks are assumed to be independent of each other and do not communicate. In cases where they do, a point-to-point link is usually assumed between all nodes. In the case where routing is involved, the intermediate nodes only introduce delays, but are not affected otherwise. On the other hand, the WSN applications studied by us consist entirely of communicating tasks. This communication of data between tasks on different nodes in a WSN affects other nodes in the system as well, since the nodes involved in routing also spend energy in the process.

3. In cases where routing is involved, classical task-mapping algorithms either have full control over routing, or assume a specific routing. Since our goal is for the macroprogramming framework to be modular, our techniques do not assume a certain routing protocol. Instead, our modeling framework allows an interface for developer to specify certain facts about the routing protocol being used.

4. While the most common constraint in traditional parallel and distributed systems is *latency*, i.e., the time taken for the tasks to complete execution, most sensor networking applications are designed to sense-and-respond for long periods of time. On the other hand, metrics such as *system life time* and *energy spent* at the node and system level are much more important in WSNs. We focus on two measures of energy-efficiency in this paper.

Additionally, although the initial information (positions, energy levels) about the target nodes is known, during the lifetime of the WSN, changing conditions, either external (variations in the environment) or internal (nodes running out of energy) may change the circumstances. Our algorithms do not address these unpredictable situations, and instead aim to provide a “good” initial mapping of tasks. We assume that during the lifetime of the system, remapping of tasks will occur to face these circumstances, for example, a distributed task-remapping algorithm can be triggered when the energy at any node goes below a certain fraction of its initial energy level. Our work attempts to utilize the global knowledge available at compile-time to obtain efficient results.

5.2 Problem Formulation

5.2.1 Motivation

As an example of data-driven macroprogramming representation, consider the following (simple) application – A room is instrumented with six wireless nodes, with three nodes equipped with temperature sensors, and two nodes connected to actuators for controlling the temperature of the room. We need to periodically determine the average temperature in the room, compare it with a threshold, and produce the corresponding actuation. One way of designing such an application at a high-level using a data-driven approach is shown in the top part of Figure 5.1 (Note that the task graph for complex applications can be an arbitrary directed acyclic graphs). Tasks T_1, T_2 and T_3 are temperature sampling tasks, which fire at a rates of f_1, f_2, f_3 and generate ambient temperature readings of size s_{14}, s_{24}, s_{34}. Task T_4 calculates the average of these readings and feeds it to T_5, which determines the action to be taken. Tasks T_6 and T_7 act upon the data generated by T_5, and control the actuators. The system for which this application is being designed is shown in the lower part of the same figure. The nodes equipped with temperature sensors are marked with a **T**, while the ones equipped with actuators are marked with an **A**. The mapping of tasks T_1 through T_7 onto the nodes of the target network is an instance of the problem faced while compiling data-driven

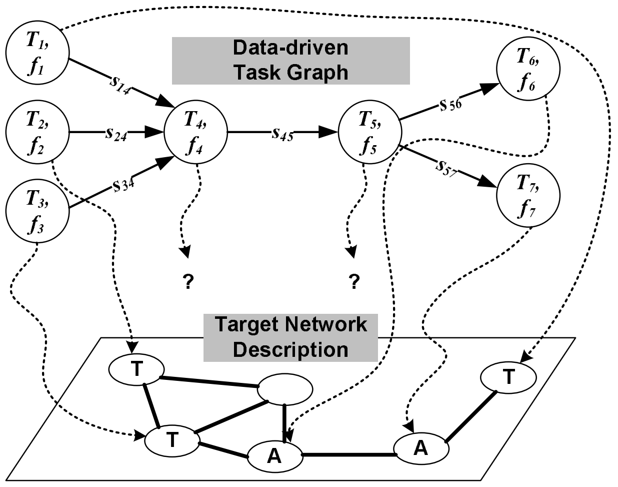


Figure 5.1: Temperature management - Task graph and target network description.

macroprograms for WSNs. The placement of the sensing tasks (T_1, T_2, T_3) and the actuating tasks(T_6 and T_7) are pre-determined to the nodes with the relevant capabilities. This fact is shown using curved broken lines in the figure. However, tasks T_4 and T_5 can be placed on any of the nodes in the floor, thus allowing for optimizations in this process.

Our aim is to capture the following aspects of the problem in a mathematical formulation:

- The data flow between tasks
- The different firing rates of the tasks
- The placement constraints of tasks onto system nodes
- The heterogeneity between the system nodes, both in terms of their initial energy capacities, as well as their ability to host certain tasks
- The heterogeneity between the various network links in the target system, in terms of energy spent per unit of data transmitted

- The energy spent at the nodes during sensing, computation, and communication.

5.2.2 Application and System Model

A **Network Description** N represents the target system of physical nodes where the WSN application is to be deployed. Each node k ($k = 1, \ldots, n$) has the following properties:

- its initial energy reserve e_k^0. We assume that the system operates in *rounds*, and denote the energy remaining at node k after t rounds by e_k^t. A **round** is defined as the least time-period after which the system behavior repeats itself.

A **Data-driven Task** i represents the sensing, processing or actuation activity in a WSN, with the following properties:

- its firing rate f_i, denoting the number of times it is invoked in one round. For tasks that are not necessarily invoked in a regular manner, developers can determine the firing rates using probabilistic estimates.

A **Data-driven Task Graph** $D = (DT, DE)$ is a directed acyclic graph (DAG) consisting of the following:

- A set $DT = \{1, \ldots, i, \ldots, m\}$ of data-driven tasks.
- A set $DE \subseteq DT \times DT$ of edges. Each edge (i, j) is labeled with the size s_{ij} of the data that task i produces for task j upon each invocation.

The **Task Execution Energy Matrix** $\mathcal{T}$ is an $m \times n$ matrix, where $\mathcal{T}_{ik}$ denotes the energy spent by node k per invocation of task i, if i is mapped onto node k. $\mathcal{T}$ can also be used to specify placement constraints as in the figure above, by setting the value of $\mathcal{T}_{ik}$ to ∞ in cases where task i cannot be instantiated on node k.

The **Routing Energy Cost Matrix** $\mathcal{R}$ for N is a $n \times n \times n$ matrix, with $\mathcal{R}_{\beta\gamma k}$ denoting the energy consumed per unit of data at node k while routing messages from node β to γ. Since the task mapping algorithms do not control the routing of data between nodes, $\mathcal{R}$ provides an estimate of the energy spent in routing.

The **Task Mapping** is a function $M : DT \to N$, designating task i to be placed on node $M(i)$.

5.2.3 Energy Costs

In a sensor network, the *cost* that developers are largely concerned with is the *energy spent* by the nodes as the system operates. We therefore use the terms *cost* to mean the energy spent at a node throughout this chapter, unless otherwise stated. Using the model defined above, we compute the following costs[1].

Computation Cost: At each node $k \in N$, the computation cost in each round is given by

$$C^k_{\text{comp}} = \sum_{i:M(i)=k} f_i \cdot \mathcal{T}_{ik} \tag{5.1}$$

Communication Cost: At each node $k \in N$, the energy spent in communicating messages in each round is given by

$$C^k_{\text{comm}} = \sum_{(i,j)\in DE} f_i \cdot s_{ij} \cdot \mathcal{R}_{M(i)M(j)k} \tag{5.2}$$

Using these node-level costs, complex system-level metrics can be represented, as discussed below.

[1]Note that the cost of sensing is included in the T_{ik} of the sensing tasks.

5.2.4 Performance Metrics

In this chapter, we illustrate the use of above modeling framework to optimize two performance metrics. The first is *energy balance*, which we consider to be achieved when the maximum fraction of energy spent by any node in the system is minimized. In other words,

$$\text{OPT}_1 = \min_{\text{all Mappings } M} \; \max_{k \in N} \frac{1}{e_o^k} \cdot (C_{\text{comp}}^k + C_{\text{comm}}^k) \tag{5.3}$$

For systems designed using data-driven macroprogramming, we can assume that the system undergoes a reconfiguration, resulting in re-computation of the task mapping, and migration of tasks, once the current energy e_k^t of any node k goes below a fraction $\alpha(0 < \alpha < 1)$ of its initial energy e_k^0. The time when this happens is called the **Time to Reconfiguration** (TTR) for the task mapping on the sensor network. Since we assume that the system behavior repeats itself in each round, OPT$_1$ also maximizes the TTR.

The second performance goal we model using our framework is the more classical *total energy spent* in the entire system. Although we believe that energy balance is a better metric to measure the quality of task placement, we use the goal of minimizing the total energy spent in the system to illustrate the modeling power of our framework. In other words,

$$\text{OPT}_2 = \min_{\text{all Mappings } M} \sum_{k \in N} (C_{\text{comp}}^k + C_{\text{comm}}^k) \tag{5.4}$$

For each of the two metrics, a *feasible* solution is possible only when all nodes have non-zero energy left at the end of one round. If there are no mappings possible for which this holds, the task-mapping algorithms should report failure. In addition to the above, our framework can be used to model other application scenarios also, e.g.

when multiple paths between two nodes are possible, or when the nodes are free to behave differently in each round.

5.2.5 Evaluation Criteria

One area where task graphs describing sensor network applications are different from those traditionally seen in parallel and distributed computing is their use for sensing the environment they are placed in, and reacting to it. This leads to certain commonly observed relationships between the data-rates on the edges of the task graphs. Therefore, while evaluating the algorithms for task mapping the input graph has to be carefully chosen. The worst-case analysis technique of testing algorithms against randomly generated task graphs with arbitrarily chosen inter-task data-rates may identify certain task mapping techniques as inferior, while these techniques may yield very good results in real-world WSN applications. Consequently, it is important that the task graphs used to evaluate these techniques are drawn from actual WSN applications.

5.3 Mathematical Formulations for Task Mapping on WSNs

5.3.1 Mixed Integer Programming Formulation for OPT$_1$

To formulate the problem as a mixed integer programming (MIP) problem, we represent task mapping M by an *assignment matrix* X, where x_{ik} is 1 if task i is assigned to node k, and 0 otherwise.

The problem can then be defined as:

Inputs:

- $D = (DT, DE)$: Data-driven Task Graph
- f_i: Firing rate for task i
- s_{ij}: Size of data transferred from task i to j on each invocation of i
- N: Network description
- $\mathcal{T}$: Task execution energy matrix
- $\mathcal{R}$: Routing energy cost matrix

Outputs:

- X: Assignment Matrix. x_{ik} is binary.

Optimization Goal:

$$\text{minimize } c$$

Constraints:

$$\sum_{k=1}^{n} x_{ik} = 1 \text{ for } i = 1, 2, \ldots, m \tag{5.5}$$

$$\frac{1}{e_k^0}\left(\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{(i,j)\in DE} \sum_{\beta=1}^{n} \sum_{\gamma=1}^{n} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot \mathcal{R}_{\beta\gamma k}\right) \leq c$$
$$\text{for } k = 1, 2, \ldots, n \tag{5.6}$$

$$x_{ik} \in \{0, 1\} \text{ for } (i, k) = (1, 1), \ldots, (m, n) \tag{5.7}$$

$$0 \leq c \leq 1 \tag{5.8}$$

The summation terms in (5.6) denote C^k_{comp} and C^k_{comm} respectively. The final constraint ensures that the MIP fails if no feasible solution exists. Note that the above is an MIP since c is real whereas x_{ik} are binary integers. Also, it is not a linear program since product terms $x_{i\beta} \cdot x_{j\gamma}$ appear in the constraints.

The above problem can be converted to a linear MIP by replacing each $x_{i\beta} \cdot x_{j\gamma}$ term with a binary variable $y_{i\beta j\gamma}$, and adding the following constraints:

$$y_{i\beta j\gamma} - x_{i\beta} \leq 0 \tag{5.9}$$

$$y_{i\beta j\gamma} - x_{j\gamma} \leq 0 \tag{5.10}$$

$$x_{i\beta} + x_{j\gamma} - y_{i\beta j\gamma} \leq 1 \tag{5.11}$$

This linearization techniques is derived from [74]. Intuitively, constraint (5.9) denotes that if edge (i, j) is mapped to path $(\beta \rightarrow \gamma)$, then task i is mapped to node β. Similarly, (5.10) denotes the constraint that if (i, j) is mapped to $(\beta \rightarrow \gamma)$, then task j is mapped to node γ. Finally, (5.11) denotes the condition that if task i is mapped to node β, and task j is mapped to node γ, then (i, j) is mapped to $(\beta \rightarrow \gamma)$.

5.3.2 MIP Formulation for OPT$_2$

Using our formulation, the objective of solving the problem to minimize the *total energy* spent by the system can be formulated as follows:

Inputs:

- $D = (DT, DE)$: Data-driven Task Graph
- f_i: Firing rate for task i
- s_{ij}: Size of data transferred from task i to j on each invocation of i
- N: Network description

- $\mathcal{T}$: Task execution energy matrix
- $\mathcal{R}$: Routing energy cost matrix

Outputs:

- X: Assignment Matrix. x_{ik} is binary.

Optimization Goal:

$$\text{minimize} \sum_{k=1}^{n}(\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{(i,j)\in DE} \sum_{\beta=1}^{n}\sum_{\gamma=1}^{n} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot \mathcal{R}_{\beta\gamma k})$$

Constraints:

$$\sum_{k=1}^{n} x_{ik} = 1 \text{ for } i = 1, \ldots, m \tag{5.12}$$

$$\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{(i,j)\in DE} \sum_{\beta=1}^{n}\sum_{\gamma=1}^{n} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot \mathcal{R}_{\beta\gamma k} \leq e_0^k \quad \text{for } k = 1, \ldots, n \tag{5.13}$$

$$x_{ik} \in \{0, 1\} \text{ for } (i, k) = (1, 1), \ldots, (m, n) \tag{5.14}$$

The above also can be converted to a MIP with linear constraints using the techniques discussed above.

5.4 Heuristic for Task Mapping

5.4.1 Greedy Algorithms for Task Mapping

Although the MIP formulation leads to optimal results, solving an MIP can be quite time consuming in practice. Our greedy heuristic for the goal of minimizing the maximum fraction of energy spent at a node (OPT$_1$) is detailed in Algorithm 5.2. The main intuition is that the algorithm sorts the edges in the task graph in non-increasing order of the traffic going on them, and then tries to map the still unmapped endpoints of each edge (i, j) so as to achieve the minimum increase in the objective function.

```
Input: $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n]$
Output: $M[m]$: Task Assignment
 1: Initialize $M[i] = -1$ for $i = \{1, \ldots, m\}$
 2: Sort $(i, j) \in DE$ in non-increasing order of $f[i] \cdot s[i][j]$
 3: for all (sorted) $(i, j)$ in $DE$ do
 4:   $minmaxCost = \infty$; $minPath = (-1, -1)$ // Initialize $minmaxCost$ and
      $minPath$ for this iteration
 5:   for all $(\alpha, \beta)$ such that $(i, j)$ can be assigned to them do
 6:     $M[i] = \alpha$, $M[j] = \beta$ // Temporarily assign $(i, j)$ to $(\alpha \rightarrow \beta)$
 7:     $maxCost = maxCost(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M)$
 8:     if $maxCost < minmaxCost$ then
 9:       $minmaxCost = maxCost$; $minPath = (\alpha, \beta)$ // Update $minmaxCost$
          and $minPath$
10:     end if
11:   end for
12:   if $minmaxCost > 1$ then
13:     declare failure. stop. // Checking for feasibility
14:   end if
15:   $M[i] = minPath.\alpha$; $M[j] = minPath.\beta$
16: end for
17: return $M$
```

Figure 5.2: *GreedyMinMax*: for OPT$_1$

Computational Complexity: Each invocation of *maxCost* takes $\theta(n(m+|DE|))$ time. During Algorithm 5.2, the sorting takes O$(|DE| \log(|DE|))$ time, and the main loops

```
Input: $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n], M[m]$
Output: *maxCost*: Maximum fraction of energy spent at any node
1: $maxCost = 0$ // Initialize max cost
2: for all $k \in N$ do
3:     $cost = 0$ // Initialize node cost
4:     for all $i \in DT$ do
5:         if $M[i] == k$ then
6:             $cost = cost + f[i] \cdot \mathcal{T}[i][k]$ // Increment computation cost
7:         end if
8:     end for
9:     for all $(i, j) \in DE$ do
10:        if $M[i] \neq 1$ AND $M[j] \neq 1$ then
11:            $cost = cost + f[i] \cdot s[i][j] \cdot \mathcal{R}[M[i]][M[j]][k]$ // Increment communication
               cost
12:        end if
13:    end for
14:    if $cost/e_0[k] > maxCost$ then
15:        $maxCost = cost/e_0[k]$
16:    end if
17: end for
18: return $maxCost$
```

Figure 5.3: *maxCost*: determining the maximum fraction of energy spent at a node

invokes Algorithm 5.3 for evaluating the *maxCost* O($|DE|n^2$) times. The total time complexity of the algorithm is O($|DE|(\log(|DE|)+n^3(m+|DE|))$). Since $|DE| > m$ in a DAG and $|DE| > \log(|DE|)$, this can be simplified to O($n^3|DE|^2$).

Algorithm 5.4 shows our modification to Algorithm 5.2 for mapping tasks for OPT_2. The algorithm calls *totalCost* subroutine (shown in Algorithm 5.5) repeatedly to determine the current total cost of the assignment, and chooses the end points of the next edge so as to minimize the total cost. Owing to the similarity in structure, its computational complexity is also O($n^3|DE|^2$).

```
**Input:** $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n]$
**Output:** $M[m]$: Task Assignment
  Initialize $M[i] = -1$ for $i = \{1, \ldots, m\}$
  Sort $(i, j) \in DE$ in non-increasing order of $f[i] \cdot s[i][j]$
  **for all** (sorted) $(i, j)$ in $DE$ **do**
    $mintotalCost = \infty$ // Initialize mintotalCost for this iteration
    $minPath = (-1, -1)$
    **for all** $(\alpha, \beta)$ such that $(i, j)$ can be assigned to them **do**
      $M[i]$ = $\alpha$
      $M[j]$ = $\beta$ // Temporarily assign $(i, j)$ to $(\alpha \to \beta)$
      $totalCost = totalCost(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M)$
      **if** $totalCost < mintotalCost$ **then**
        $mintotalCost = totalCost$ // Update mintotalCost
        $minPath = (\alpha, \beta)$
      **end if**
    **end for**
    $maxCost = maxCost(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M)$
    **if** $maxCost > 1$ **then**
      **declare failure. stop.** // Checking for feasibility
    **end if**
    $M[i] = minPath.\alpha$
    $M[j] = minPath.\beta$
  **end for**
  **return** $M$
```

Figure 5.4: *GreedyMinTotal*: for OPT$_2$

5.4.2 Worst-case Analysis

Since both *GreedyMinMax* and *GreedyMinTotal* are heuristics, we explored the situations when they can give sub-optimal results. We introduce the notion of the *cost of an algorithm* for this purpose – the cost of *GreedyMinMax* is defined a the maximum fraction of energy spent in one round at any node in N, and the cost of *GreedyMinTotal* is the total energy spent by all the nodes in N in one round, when tasks are mapped according to the heuristic.

Theorem 1: For any integer $\upsilon \geq 1$, there are problem instances for which the cost of *GreedyMinMax* (*GreedyMinTotal*) is arbitrarily close to $\upsilon \times$ OPT$_1$ ($\upsilon \times$ OPT$_2$).

```
Input: $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n], M[m]$
Output: totalCost: Total energy spent by nodes in $N$
  $totalCost = 0$ // Initialize total cost
  for all $k \in N$ do
    $cost = 0$; // Initialize node cost
    for all $i \in DT$ do
      if $M[i] == k$ then
        $cost = cost + f[i] \cdot \mathcal{T}[i][k]$ // Increment computation cost
      end if
    end for
    for all $(i, j) \in DE$ do
      if $M[i] \neq 1$ AND $M[j] \neq 1$ then
        $cost = cost + f[i] \cdot s[i][j] \cdot \mathcal{R}[M[i]][M[j]][k]$ // Increment communication
        cost
      end if
    end for
    $totalCost = totalCost + cost$
  end for
  return $totalCost$
```

Figure 5.5: *totalCost*: determining the total energy spent in the system

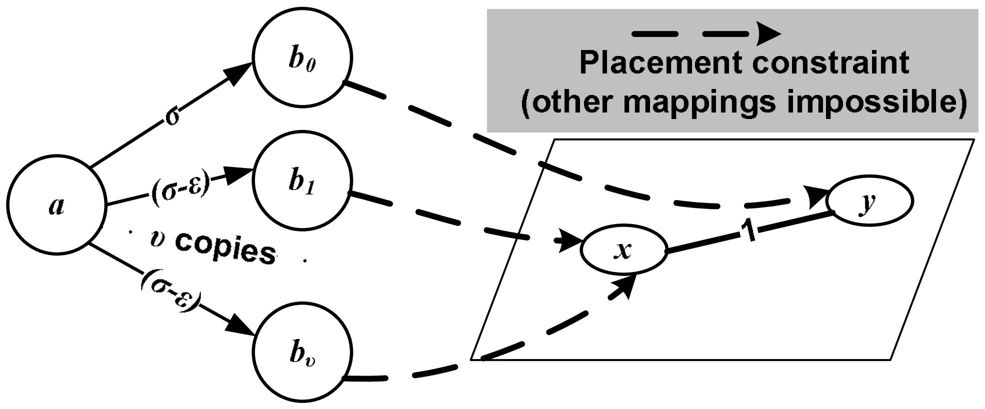


Figure 5.6: Worst-case scenario for *GreedyMinMax* and *GreedyMinTotal*

Proof: Consider a situation as illustrated in Figure 5.6. $T_{ax} = T_{ay} = 0$, and the other tasks can only be placed on the nodes indicated by the arrows. Let us also assume that $f_a = 1$, $e_0^x = e_0^y = e_0$, and both nodes in N spend one unit of energy per unit of data transmitted on the link between them. Finally, $e_0 \gg k \gg \varepsilon > 0$. The optimal solution, both for OPT_1 and OPT_2, is to place a on node x, thereby causing only the data on the (a, b_0) edge in DE to go on the network, costing k units of energy to be spent by node x (and the entire system) in each round. The greedy algorithms, however, start with placing the costliest edge (a, b_0) in the best possible manner, co-locating a and b_0 on node y. This leads to $\upsilon \times (k - \varepsilon)$ traffic to go over the $y \rightarrow x$ link.

$$\text{We thus get:} \quad \text{OPT}_1 = \frac{1}{e_0}\sigma \tag{5.15}$$

$$\Rightarrow \text{cost}(GreedyMinMax) = \frac{1}{e_0}\upsilon \times (\sigma - \varepsilon) \approx \upsilon \times \text{OPT}_1 \tag{5.16}$$

$$\text{Similarly, } \text{OPT}_2 = 2\sigma \tag{5.17}$$

$$\Rightarrow \text{cost}(GreedyMinTotal) = 2\upsilon \times (\sigma - \varepsilon) \approx \upsilon \times \text{OPT}_2 \tag{5.18}$$

hence proving the theorem. ■

Theorem 2: There are problem instances for which *GreedyMinMax* and *GreedyMinTotal* will terminate in failure although a feasible solution exists.

Proof: Consider the situation as illustrated in Figure 5.6. However, in this case, assume that $e_0 = k \gg \varepsilon > 0$. The optimal solution (given by the MIP formulation) will still place task a on node y, while the Greedy algorithms will try to place it on node y. Note that for $\upsilon \geq 2$, this will lead to an infeasible solution, as the nodes end up spending $> e_0$ energy. The proof follows. ■

5.5 Task Mapping with Multi-Path Routing

In many WSN applications, multiple routes are possible between a pair of nodes. In this section, we provide generalized versions of our problem formulations to incorporate this condition. For this purpose, the following changes are made to the model proposed in Section 5.2:

- We assume that a constant Φ number of paths are possible to be taken between any pair of nodes β and γ in N.
- We further assume that for each pair of communicating tasks (i, j) mapped to nodes β and γ respectively, one of the Φ $\beta \rightarrow \gamma$ paths (say ρ) is chosen. Note that for another pair of communicating tasks (s, t) mapped to β and γ, another $\beta \rightarrow \gamma$ path ρ' can be chosen.
- To incorporate the above, we redefine the *routing energy cost matrix* $\mathcal{R}$ to be a $n \times n \times n \times \Phi$ matrix, with $\mathcal{R}_{\beta\gamma k\rho}$ denoting the energy consumed per unit of data at node k while routing messages from node β to γ, using the *ρth* routing option.
- The task-mapping algorithms, apart from determining the task mapping M, also need to provide the **routing path choice mapping** $P : DE \rightarrow \{1, 2, \ldots, \Phi\}$.
- The communication cost at node k is now given by:

$$C^k_{\text{comm}} = \sum_{e(=(i,j))\in DE} f_i \cdot s_{ij} \cdot \mathcal{R}_{M(i)M(j)kP(e)} \tag{5.19}$$

5.5.1 MIP Formulation for OPT$_1$ when Multi-Path Routing is Possible

The problem of task-mapping and route choice to minimize the maximum fraction of energy spent at a node can thus be formulated as:

Inputs:

- $D = (DT, DE)$: Data-driven Task Graph
- f_i: Firing rate for task i
- s_{ij}: Size of data transferred from task i to j on each invocation of i
- N: Network description
- $\mathcal{T}$: Task execution energy matrix
- $\mathcal{R}$: Routing energy cost matrix, as modified above.

Outputs:

- X: Assignment Matrix. x_{ik} is binary.
- Z: Routing Path Choice Matrix. $z_{e\rho}$ is binary, and is 1 if the traffic over edge e in DE is routed along the ρth path.

Optimization Goal:

$$\text{minimize } \ c$$

Constraints:

$$\sum_{k=1}^{n} x_{ik} = 1 \ \text{ for } i = 1, 2, \ldots, m \tag{5.20}$$

$$\sum_{\rho=1}^{\Phi} z_{e\rho} = 1 \text{ for each } e \tag{5.21}$$

$$\frac{1}{e_k^0}(\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{e=(i,j)\in DE} \sum_{\beta=1}^{n} \sum_{\gamma=1}^{n} \sum_{\rho=1}^{C} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot z_{e\rho} \cdot \mathcal{R}_{\beta\gamma k\rho}) \leq c \quad \text{for } k = 1, \ldots, n \tag{5.22}$$

$$x_{ik} \in \{0,1\} \text{ for } (i,k) = (1,1), \ldots, (m,n) \tag{5.23}$$

$$z_{e\rho} \in \{0,1\} \text{ for each combination of } (e,\rho) \tag{5.24}$$

$$0 \leq c \leq 1 \tag{5.25}$$

Note that the above is an MIP since c is real whereas x_{ik} and $z_{e\rho}$ are binary integers. Also, it is not a linear program since product terms $x_{i\beta} \cdot x_{j\gamma} \cdot z_{e\rho}$ appear in the constraints.

The above problem can be converted to a linear MIP by repeatedly applying the techniques discussed in the previous section. We first absorb each $(x_{i\beta}, x_{j\gamma})$ pair into a variable $y_{i\beta j\gamma}$, and then introduce another set of variables, one to absorb each $(y_{i\beta j\gamma}, z_{e\rho})$ pair, to get the following:

$$y_{i\beta j\gamma} - x_{i\beta} \leq 0 \tag{5.26}$$

$$y_{i\beta j\gamma} - x_{j\gamma} \leq 0 \tag{5.27}$$

$$x_{i\beta} + x_{j\gamma} - y_{i\beta j\gamma} \leq 1 \tag{5.28}$$

$$u_{i\beta j\gamma\rho} - y_{i\beta j\gamma} \leq 0 \tag{5.29}$$

$$u_{i\beta j\gamma\rho} - z_{e\rho} \leq 0 \tag{5.30}$$

$$y_{i\beta j\gamma} + z_{e\rho} - u_{i\beta j\gamma\rho} \leq 1 \tag{5.31}$$

5.5.2 MIP Formulation for OPT$_2$ when Multi-Path Routing is Possible

Using our formulation, the objective of solving the problem to minimize the *total energy* spent by the system can be formulated as follows:

Inputs:

- $D = (DT, DE)$: Data-driven Task Graph
- f_i: Firing rate for task i
- s_{ij}: Size of data transferred from task i to j on each invocation of i
- N: Network description
- $\mathcal{T}$: Task execution energy matrix
- $\mathcal{R}$: Routing energy cost matrix, as modified above.

Outputs:

- X: Assignment Matrix. x_{ik} is binary.
- Z: Routing Path Choice Matrix. $z_{e\rho}$ is binary.

Optimization Goal:

$$\text{minimize} \sum_{k=1}^{n} (\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{e=(i,j)\in DE} \sum_{\beta=1}^{n} \sum_{\gamma=1}^{n} \sum_{\rho=1}^{\Phi} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot z_{e\rho} \cdot \mathcal{R}_{\beta\gamma k\rho})$$

Constraints:

$$\sum_{k=1}^{n} x_{ik} = 1 \text{ for } i = 1, 2, \ldots, m \tag{5.32}$$

$$\sum_{\rho=1}^{\Phi} z_{e\rho} = 1 \text{ for each combination of } (e, \beta, \gamma) \tag{5.33}$$

$$\sum_{i=1}^{m} f_i \cdot \mathcal{T}_{ik} \cdot x_{ik} + \sum_{e=(i,j)\in DE} \sum_{\beta=1}^{n} \sum_{\gamma=1}^{n} \sum_{\rho=1}^{\Phi} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot z_{e\rho} \cdot \mathcal{R}_{\beta\gamma k\rho} \le e_0^k$$

$$\text{for } k = 1, \ldots, n \tag{5.34}$$

$$x_{ik} \in \{0, 1\} \text{ for } (i, k) = (1, 1), \ldots, (m, n) \tag{5.35}$$

$$z_{e\rho} \in \{0, 1\} \text{ for each combination of } e, \beta, \gamma, \rho \tag{5.36}$$

Note that the above can be converted to a MIP with linear constraints using the linearization techniques used by us.

5.5.3 Greedy Heuristics for Task-mapping with Multi-Path Routing

In view of the changed system model, we can modify the algorithms proposed in Section 5.4. Our greedy heuristic for the goal of minimizing the maximum fraction of energy spent at a node (OPT$_1$) is detailed in Algorithm 5.7. The main intuition is that the algorithm sorts the edges in the task graph in non-increasing order of the traffic going on them, and then tries to map the still unmapped endpoints of each edge (i, j) and determine the best route to be taken by the data items transferred between i and j them, so as to achieve the minimum increase in the objective function.

Computational Complexity: Each invocation of *maxCostM* takes $\theta(n(m + |DE|))$ time. During Algorithm 5.7, the sorting takes O$(|DE| \log(|DE|))$ time, and the main

```
Input: $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n][\Phi], f[m], s[m][m], e_o[n]$
Output: $M[m]$: Task Assignment, $P[|DE|]$: Routing Path Choice
  Initialize $M[i] = -1$ for $i = \{1, \ldots, m\}$
  Initialize all entries $P[e] = -1$
  Sort $(i, j) \in DE$ in non-increasing order of $f[i] \cdot s[i][j]$
  for all (sorted) $e = (i, j)$ in $DE$ do
    $minmaxCost = \infty$ // Initialize minmaxCost for this iteration
    $minPath = (-1, -1)$
    for all $(\alpha, \beta)$ such that $(i, j)$ can be assigned to them do
      $M[i]$ = $\alpha$
      $M[j]$ = $\beta$ // Temporarily assign $(i, j)$ to $(\alpha \to \beta)$
      for $\rho = 1$ to $\Phi$ do
        $P[e] = \rho$ // Temporarily choose the $\rho th$ routing option
        $maxCost = maxCostM(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M, P)$
        if $maxCost < minmaxCost$ then
          $minmaxCost = maxCost$ // Update mintotalCost
          $minPath = (\alpha, \beta, \rho)$
        end if
      end for
    end for
    if $minmaxCost > 1$ then
      declare failure. stop. // Checking for feasibility
    end if
    $M[i] = minPath.\alpha$
    $M[j] = minPath.\beta$
    $P[e][M[i]][M[j]] = minPath.\rho$
  end for
  return $M, P$
```

Figure 5.7: *GreedyMinMaxM*: for OPT$_1$ with Multi-Path Routing

loops invokes Algorithm 5.8 for evaluating the *maxCost* O($|DE|n^2\Phi$) times. The total time complexity of the algorithm is O($|DE|(\log(|DE|) + n^3(m + |DE|)\Phi)$). Since $|DE| > m$ in a DAG and $|DE| > \log(|DE|)$, this can be simplified to O($n^3|DE|^2\Phi$).

Algorithm 5.9 shows our modification to Algorithm 5.7 for mapping tasks for OPT_2. The algorithm calls *totalCostM* subroutine (shown in Algorithm 5.10) repeatedly to determine the current total cost of the assignment, and chooses the end points

```
Input: $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n], M[m], P[|DE|][n][n]$
Output: maxCostM: Maximum fraction of energy spent at any node
  $maxCostM = 0$ // Initialize max cost
  for all $k \in N$ do
    $cost = 0$ // Initialize node cost
    for all $i \in DT$ do
      if $M[i] == k$ then
        $cost = cost + f[i] \cdot \mathcal{T}[i][k]$ // Increment computation cost
      end if
    end for
    for all $e = (i, j) \in DE$ do
      if $M[i] \neq -1$ AND $M[j] \neq -1$ AND $P[e] \neq -1$ then
        $cost = cost + f[i] \cdot s[i][j] \cdot \mathcal{R}[M[i]][M[j]][k][P[e]]$ // Increment
        communication cost
      end if
    end for
    if $cost/e_0[k] > maxCostM$ then
      $maxCostM = cost/e_0[k]$
    end if
  end for
  return $maxCostM$
```

Figure 5.8: *maxCostM*: determining the maximum fraction of energy spent at a node

of the next edge and the path choice so as to minimize the total cost. Owing to the similarity in structure, its computational complexity is also O($n^3|DE|^2\Phi$).

Worst Case Analysis: Since *GreedyMinMaxM* and *GreedyMinTotalM* are generalized versions of the algorithms discussed in Section 5.4, the problem instance discussed in Section 5.4.2 acts as a special case of the task-mapping problem with multi-path routing, with the maximum number of routes $\Phi = 1$. Therefore, the same worst-case bounds hold for the algorithms discussed in this section also.

```
**Input:** $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n]$
**Output:** $M[m]$: Task Assignment, $P[|DE|]$: Routing Path Choice
  Initialize $M[i] = -1$ for $i = \{1, \ldots, m\}$
  Initialize all entries $P[e][\beta][\gamma] = -1$
  Sort $(i, j) \in DE$ in non-increasing order of $f[i] \cdot s[i][j]$
  **for all** (sorted) $e = (i, j)$ in $DE$ **do**
    $mintotalCost = \infty$ // Initialize mintotalCost for this iteration
    $minPath = (-1, -1)$
    **for all** $(\alpha, \beta)$ such that $(i, j)$ can be assigned to them **do**
      $M[i] = \alpha$
      $M[j] = \beta$ // Temporarily assign $(i, j)$ to $(\alpha \rightarrow \beta)$
      **for** $\rho = 1$ to $\Phi$ **do**
        $P[e] = \rho$ // Temporarily choose the $\rho th$ routing option
        $totalCost = totalCostM(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M, P)$
        **if** $totalCost < mintotalCost$ **then**
          $mintotalCost = totalCost$ // Update mintotalCost
          $minPath = (\alpha, \beta, \rho)$
        **end if**
      **end for**
    **end for**
    $maxCost = maxCostM(D, N, \mathcal{T}, \mathcal{R}, f, s, e_0, M, P)$
    **if** $maxCost > 1$ **then**
      **declare failure. stop.** // Checking for feasibility
    **end if**
    $M[i] = minPath.\alpha$
    $M[j] = minPath.\beta$
    $P[e] = minPath.\rho$
  **end for**
  **return** $M, P$
```

Figure 5.9: *GreedyMinTotalM*: for OPT$_2$ with Multi-Path Routing

5.6 Evaluation

5.6.1 Reference Applications

For evaluating our techniques, we use task-graphs derived from the two applications discussed in Section 3.3. The first is a building environment management application for monitoring heating, ventilation and air-conditioning (HVAC), similar in spirit to

```
Input:
  $D(= DT, DE), N, \mathcal{T}[m][n], \mathcal{R}[n][n][n], f[m], s[m][m], e_o[n], M[m], P[|DE|]$
Output: totalCostM: Total energy spent by nodes in $N$
  $totalCostM = 0$ // Initialize total cost
  for all $k \in N$ do
    $cost = 0$; // Initialize node cost
    for all $i \in DT$ do
      if $M[i] == k$ then
        $cost = cost + f[i] \cdot \mathcal{T}[i][k]$ // Increment computation cost
      end if
    end for
    for all $e = (i, j) \in DE$ do
      if $M[i] \neq -1$ AND $M[j] \neq -1$ AND $P[e]! = -1$ then
        $cost = cost + f[i] \cdot s[i][j] \cdot \mathcal{R}[M[i]][M[j]][k][P[e]]$ // Increment comm. cost
      end if
    end for
    $totalCostM = totalCostM + cost$
  end for
  return $totalCostM$
```

Figure 5.10: *totalCostM*: determining the total energy spent in the system

other applications in the literature [24]. We consider a set of nodes spread across a building, with each node possibly attached to a temperature sensor, a humidity sensor and an actuator that can control the temperature and humidity of a region. The aim of the system is to maintain desirable temperature and humidity levels in each room of the building, by correlating the information from the sensor installed in the room, and using it to drive actuation.

Figure 5.11 describes our application as a data-driven task graph. The *Temperature Sampler* and *Humidity Sampler* tasks – instantiated on the nodes with relevant sensors – sample their surroundings and generate temperature and humidity readings. This data is then sent to the *Collector* task, one of which is placed in each room. Upon processing the data, the *Collector* produces a command for the actuating tasks and sends the data

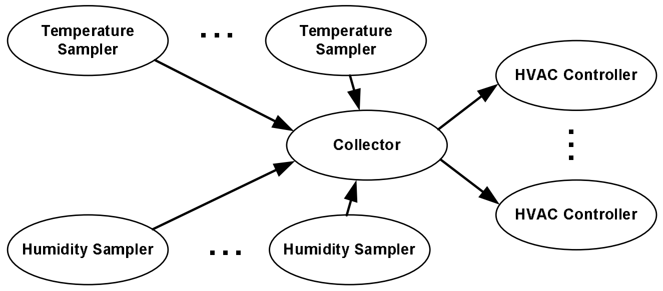


Figure 5.11: A task graph for HVAC management.

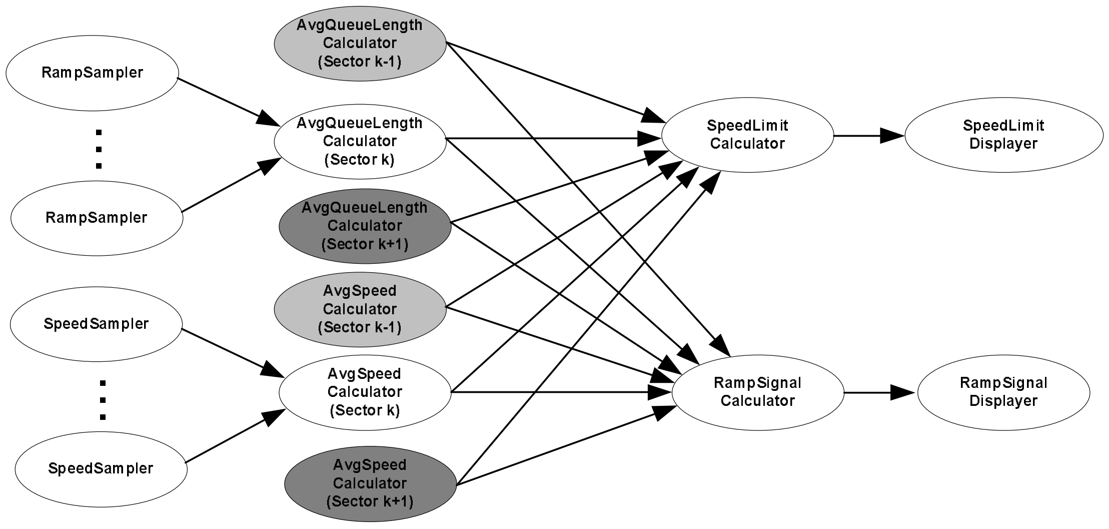

AvgQueueLength Calculator (Sector k-1)
RampSampler
AvgQueueLength Calculator (Sector k)
SpeedLimit Calculator
SpeedLimit Displayer
RampSampler
AvgQueueLength Calculator (Sector k+1)
AvgSpeed Calculator (Sector k-1)
SpeedSampler
AvgSpeed Calculator (Sector k)
RampSignal Calculator
RampSignal Displayer
SpeedSampler
AvgSpeed Calculator (Sector k+1)

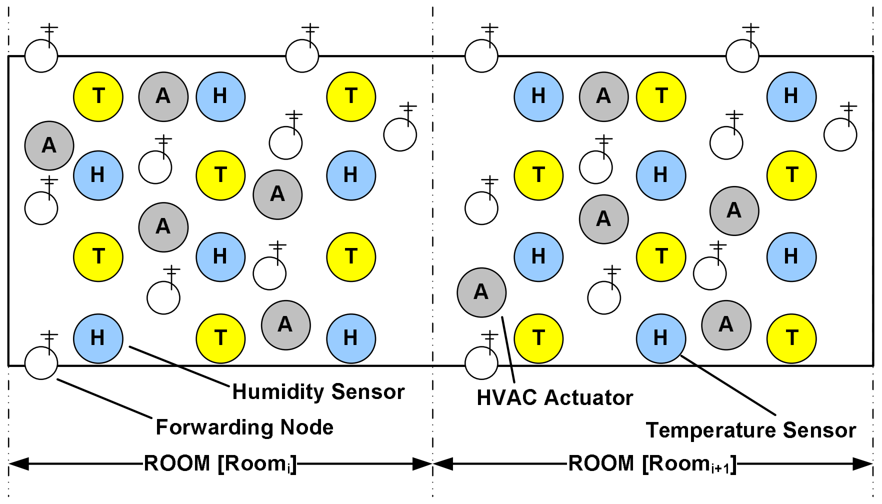
Figure 5.12: An task graph for highway traffic management.

to the *HVAC Controller* task, which is placed on all nodes with an HVAC Actuator and responds to the *Action* data item by adjusting the temperature/humidity controls.

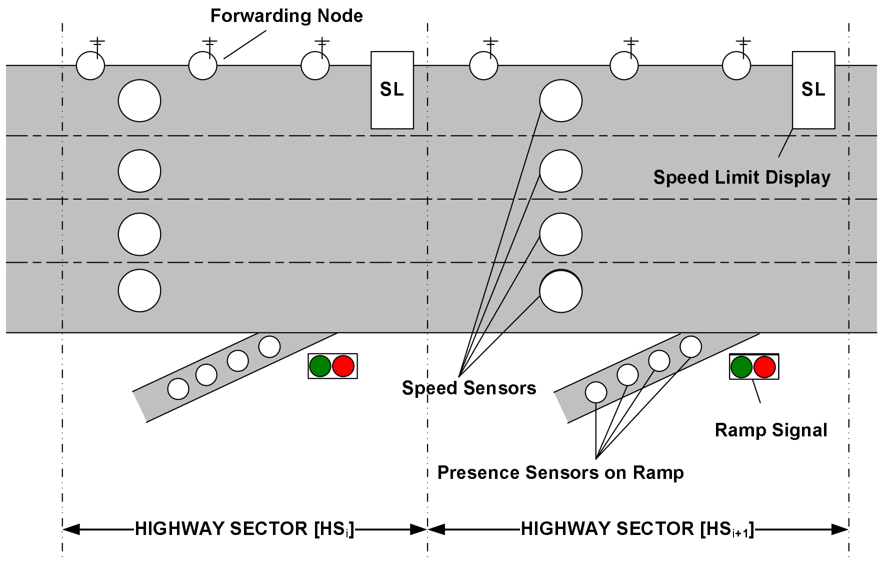
The second application, illustrated in Figure 5.12, describes a highway traffic management system, similar in spirit to [48]. In this case, two different sub-goals must be achieved - regulating the speed of vehicles on the highway by controlling speed limit displays, and controlling the access to the highway by means of red/green signals on the ramps. The highway is divided into sectors, and sensors are deployed on the highway lanes and ramps to sense the speed and presence of vehicles, respectively. The sensed data goes through a multi-stage process where it is first aggregated w.r.t. a single sector to derive an average measure (*AvgSpeedCalculator* and *AvgQueueLength-Calculator* tasks). The *SpeedLimitCalculator* and *RampSignalCalculator* tasks take the average speeds and queue lengths produced in the neighboring highway sectors (as shown in the figure), and compute the desired actions to be sent to the *SpeedLimitDisplayer* and *RampSignalDisplayer* tasks, which are located on the nodes attached to the corresponding actuators.

5.6.2 Experiments

For evaluating the relative performance of our heuristics, we applied them on the task graphs discussed in Section 5.6.1, by using our algorithms to map their tasks onto a various simulated target deployments (shown in Figure 5.13) to map the tasks onto. For the HVAC application, we placed an equal number of temperature and humidity sensors in a grid in a room, and assigned the location of the HVAC actuators randomly. We also placed extra nodes in the room for maintaining connectivity. For the traffic application, we placed forwarding nodes uniformly apart at the edge of the highway, and randomly distributed the speed sensors on the four lanes so that each of them was in range of at least another speed sensor or a forwarding node. Similarly, the presence sensors were randomly distributed on the ramp so that each of them was in range of at least one speed sensor or another presence sensor. The node controlling the ramp signals and the speed limit displays were placed between different sectors, on opposite sides of the road. Note that for both the applications, owing to the placement constraints of the applications, the number of tasks m is $O(n)$ for our experiments, where n is the number of nodes.



(a) HVAC Application



(b) Traffic Application

Figure 5.13: Node placement in reference applications.

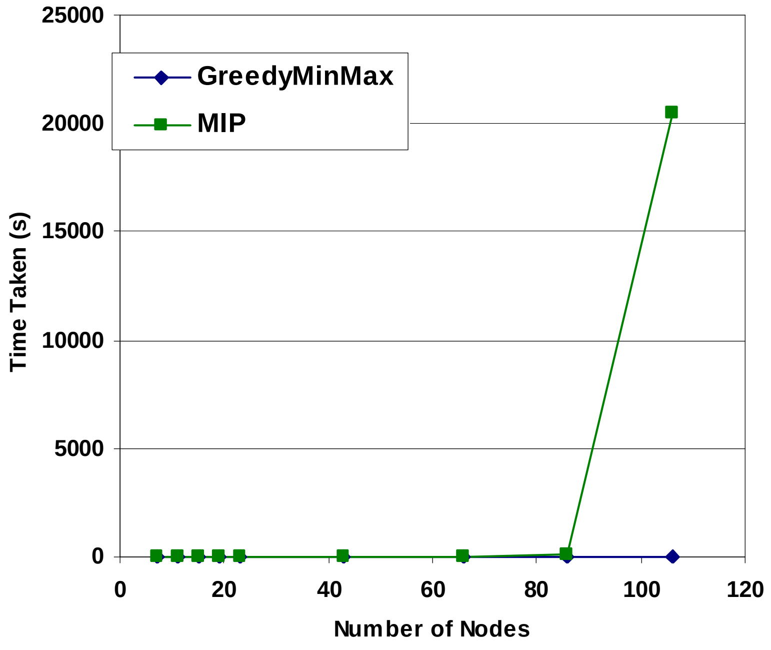
Experimental Results: In our experiments, we assumed that all nodes started with a sufficiently high initial energy level e_0. The routing energy cost matrix $\mathcal{R}$ was obtained by using a shortest path algorithm on the network, assuming equal energy spent by all nodes on a route, and all data items were assumed to be of unit size ($s_{ij} = 1$). The task execution energy matrix $\mathcal{T}$ was set up to represent placement constraints:

$\mathcal{T}_{ik} = 0$ when task i could be placed on node k, ∞ when it could not. The tasks which performed sensing and actuating were tied to a node with the relevant capabilities. Finally, the f_i for each task was computed as follows: For sensing tasks, f_i was set to 10, and for all other tasks j, f_j was set to the sum of the firing rates of tasks on the other ends of the incoming edges. This represented the fact that task j fires whenever there is data available for it. For the multi-path scenario, we generated the routing matrix using the generalized Floyd algorithm [34] with $\Phi = 3$.

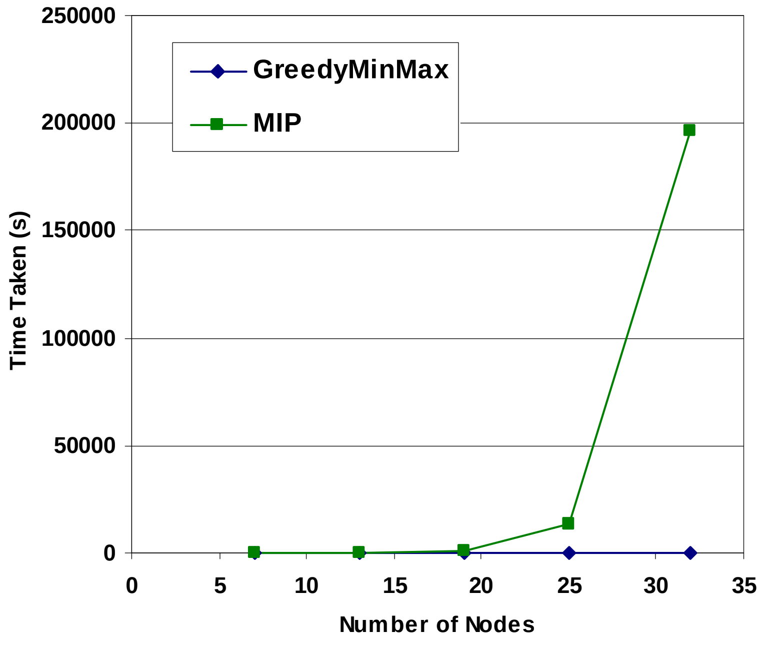
We ran our experiments on a PC with a dual-core Pentium processor running at 1.6GHz, with 2GB of RAM. We implemented our greedy algorithm in Java, and solved the MIPS using the lp_solve [62] linear programming toolkit. The time taken for computing task placements for both the applications so as to minimize the maximum fraction of energy spent by any node (OPT$_1$) is shown in Figure 5.14. The time taken by the two techniques for placing tasks so as to minimize the total energy spent in the system (OPT$_2$) is shown in Figure 5.15.

In our experiments, the Greedy algorithms obtained sub-optimal results only while computing task-mappings for minimizing total energy in some of the HVAC application. For the traffic application, and the for *all* instances of OPT$_1$ (which we believe is a better indicator of system lifetime), the solution given by the greedy algorithm was the same as the one given by the MIP. Our experiments clearly show that the greedy algorithms take much less time that the MIP formulation in finding the mappings. This showcases the efficacy of the algorithms in solving the task-mapping problem for complex real-world WSN applications.

In experiments conducted where multiple paths were possible (shown in Figures 5.16 and 5.17), we see that the time taken by the heuristics still outperforms the MIP solver in terms of time. Note that the graphs for the traffic management application denote the fact that the MIP solver did not terminate in a long time for some instances. As before, the quality of solution given by our heuristics were found to be as good as that of the MIP, with the cost of the (few) outliers not being more than 1.5 times the optimal cost.

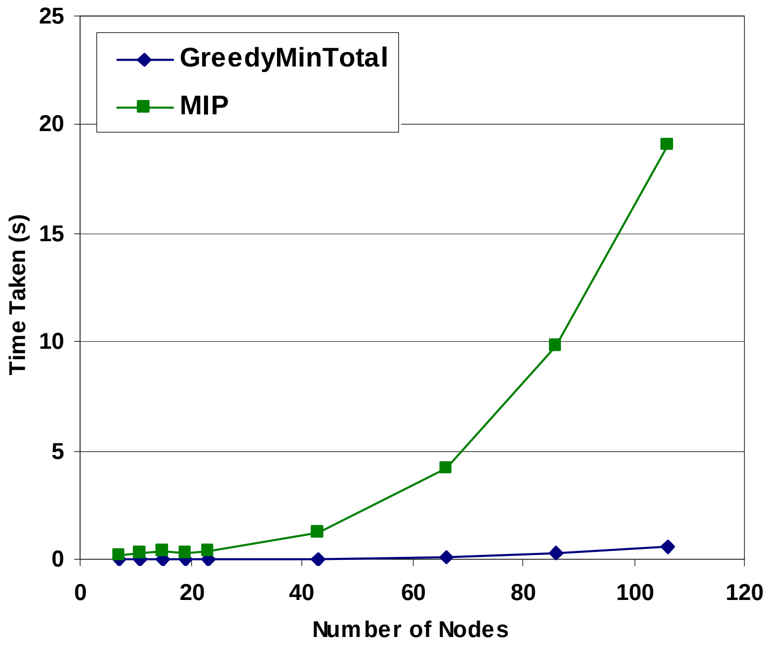


(a) HVAC Management

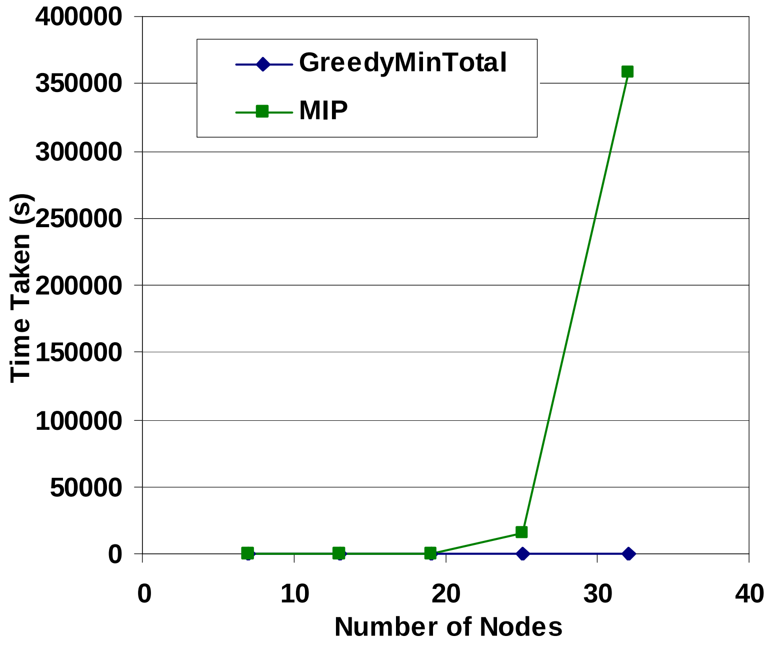


(b) Traffic Management

Figure 5.14: Task-mapping time for OPT_1

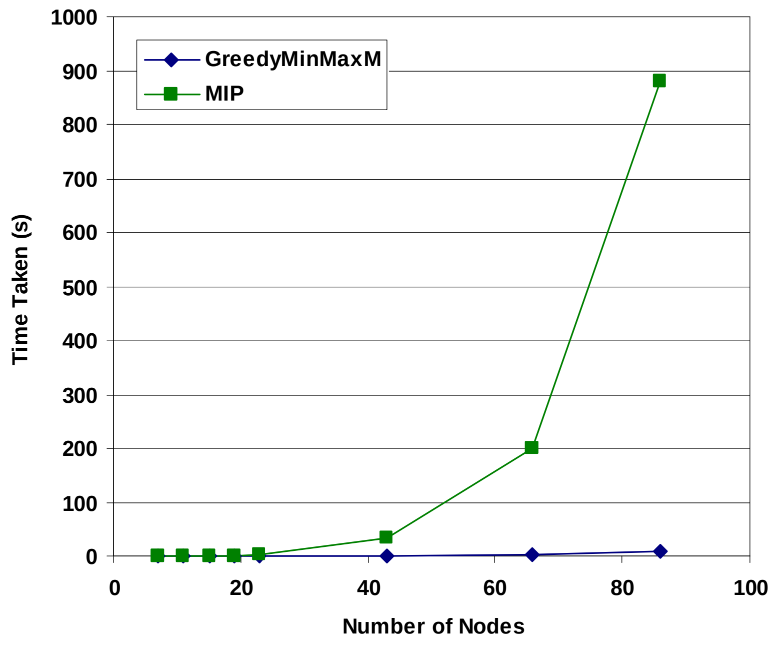


(a) HVAC Management

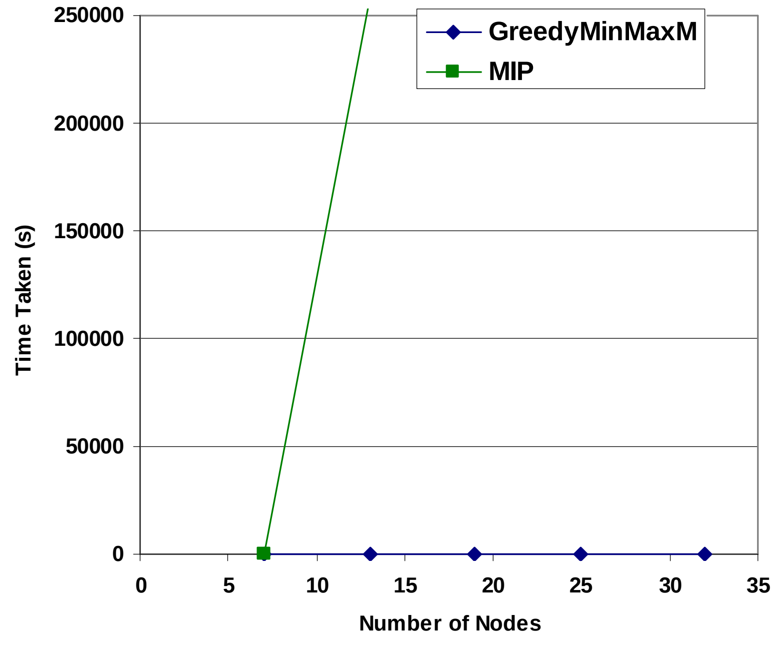


(b) Traffic Management

Figure 5.15: Task-mapping time for OPT_2

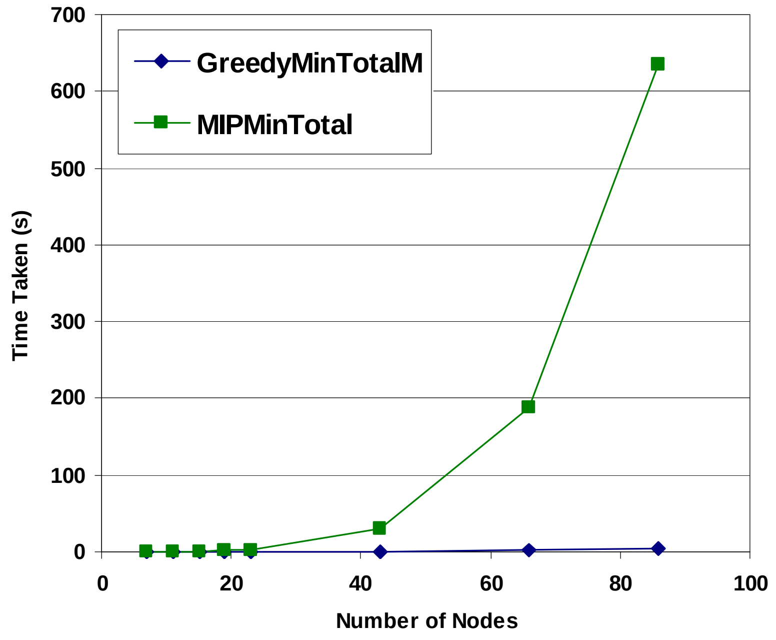


(a) HVAC Management - Multipath

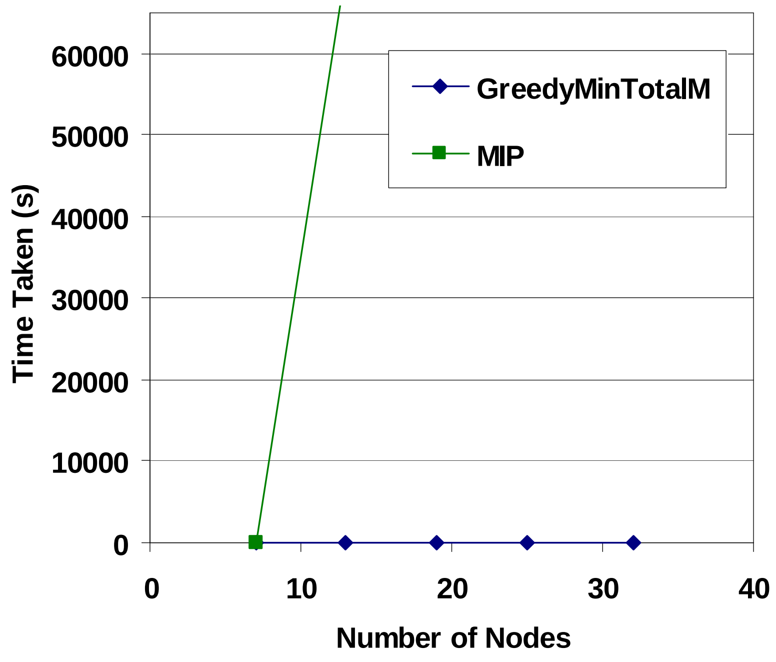


(b) Traffic Management - Multipath

Figure 5.16: Task-mapping time for OPT_1 (MultiPath)



(a) HVAC Management - MultiPath



(b) Traffic Management - Multipath

Figure 5.17: Task-mapping time for OPT_2 (MultiPath)

Chapter 6

Graphical Toolkit for WSN Application Development

6.1 *Srijan*: Graphical Toolkit for Data-driven WSN Macroprogramming

Since the goal of WSN macroprogramming research is to make application development easier for the *domain expert*, we believe that it is absolutely necessary to make *easy-to-use toolkits* for macroprogramming available to them in order to both make their task easier, as well as to gain feedback about the macroprogramming paradigms themselves. Although various efforts exist in literature for making WSN application development easier, very few general purpose graphical toolkits for macroprogramming are publicly available for the application developer to choose from. In this section, we show how we have incorporated our macroprogram compilation framework into *Srijan*, an easy-to-use graphical front-end to the various steps involved in developing an application using ATaG. Figure 6.1 shows the various components of our toolkit that the application developer can use. The clear arrows show the *inputs*, while the gray arrows show the *output* of each component. The various components of *Srijan* are as follows.

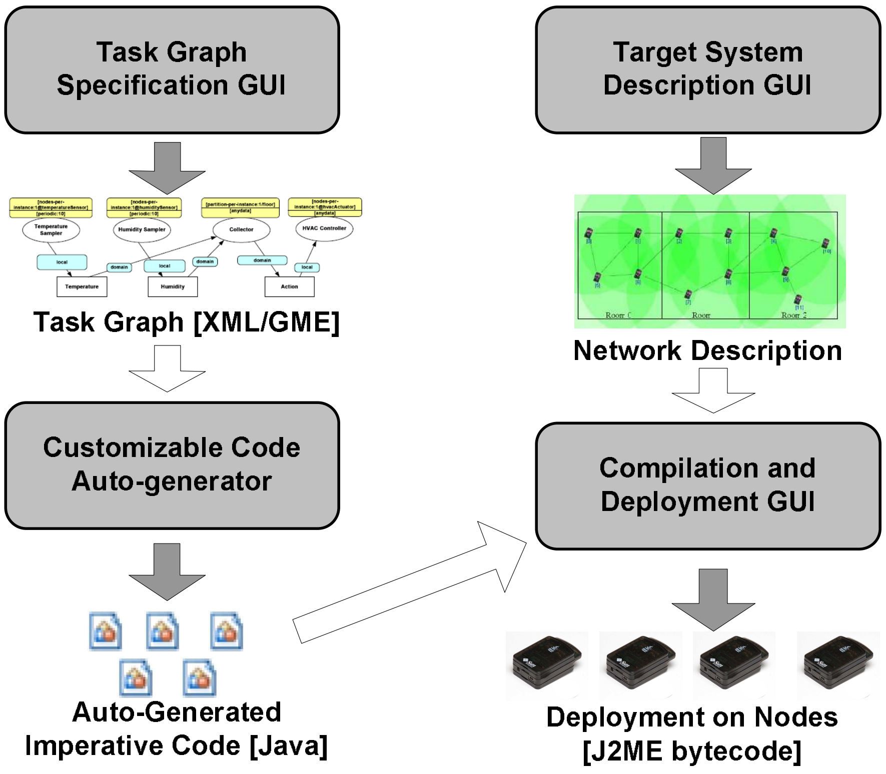
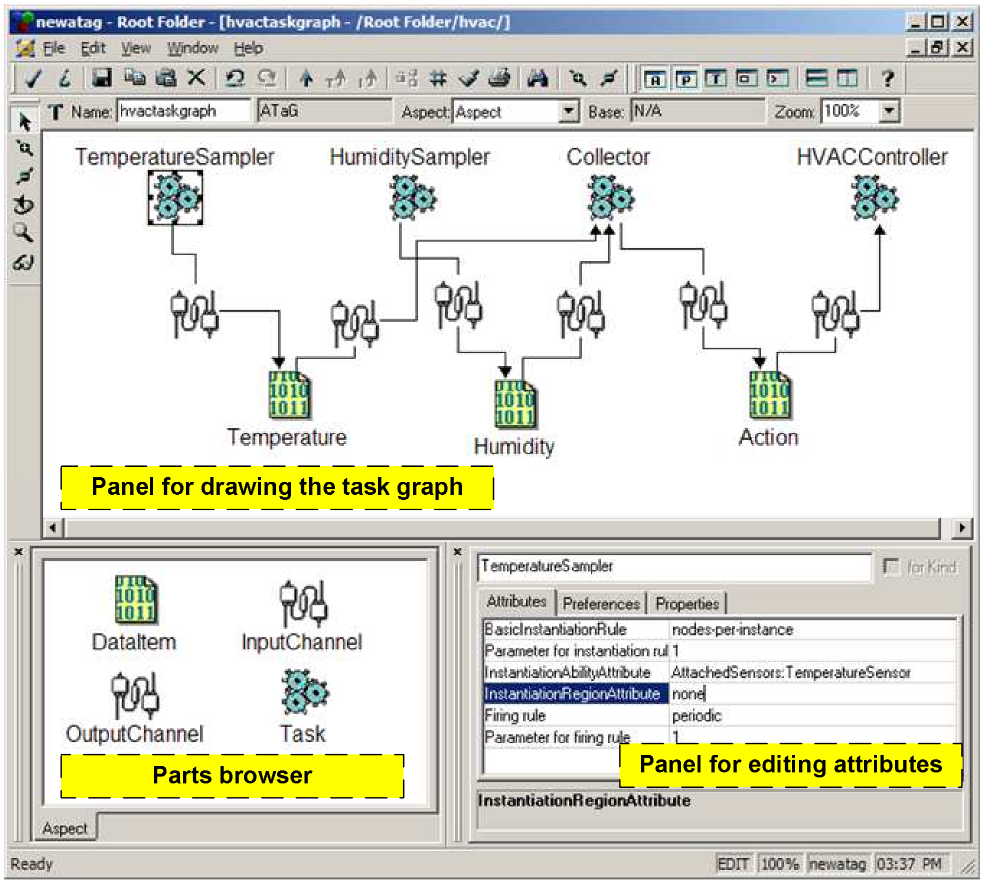

Task Graph Specification GUI
Target System Description GUI
Task Graph [XML/GME]
Network Description
Customizable Code Auto-generator
Compilation and Deployment GUI
Auto-Generated Imperative Code [Java]
Deployment on Nodes [J2ME bytecode]

Figure 6.1: Overview of application development using *Srijan*

6.1.1 Abstract Task Graph Description GUI

The ability of specifying a WSN application in a graphical manner as interconnected task and data items is a major part of ease-of-use provided by ATaG. In *Srijan*, we have customized the Generic Modeling Environment (GME) [39] for providing this facility to the application developer. Figure 6.2 shows the application introduced in Section 3.3 specified using our GUI. The developer can drag and drop *parts* representing the tasks, data items, and channels of the application from the *parts browser* onto the workspace, and draw connections between them to show their inter-relationships. The annotations of each component can then be set by clicking on it and editing the attributes in the *attribute editing panel*. Once the details of the task graph are specified, *Srijan* generates an XML file representing it by invoking the *PatternProcessor* model interpretor



Name: hvactaskgraph ATaG Aspect: Aspect Base: N/A Zoom: 100%

TemperatureSampler

BasicInstantiationRule	nodes-per-instance
Parameter for instantiation rul	1
InstantiationAbilityAttribute	AttachedSensors:TemperatureSensor
InstantiationRegionAttribute	none
Firing rule	periodic
Parameter for firing rule	1

InstantiationRegionAttribute

EDIT 100% newatag 03:37 PM

Figure 6.2: HVAC application in our task description GUI

of GME. This XML representation of the task graph, parts of which are shown in Figure 6.3, can then be used by the other components of our toolkit, discussed later in this section.

Our work towards providing the above facilities to the programmer consisted of two parts. Firstly, we developed a *metamodel* in GME for ATaG, describing the possible attributes of each component of an ATaG program, as well as the relationships between them. Secondly, we developed a *pattern file* for ATaG, which is be used by the GME pattern processor to generate a properly formatted XML file given a particular ATaG task graph.

```
<atagprogram>
  <tasklist>
    <task name="TemperatureSampler">
      <firingrule>
        <periodic period="10" />
      </firingrule>
      <location>
         <nodes-per-instance number="1" />
         <ability-attribute value="attachedSensors:TemperatureSensor"/>
         <region-attribute regionname="none"/>
      </location>
    </task>
...
    <task name="Collector">
      <firingrule>
        <anydata />
      </firingrule>
      <location>
         <partition-per-instance number="1" />
         <ability-attribute value="none"/>
         <region-attribute regionname="Room"/>
      </location>
    </task>
  </tasklist>

  <dataitemlist>
    <dataitem name="Temperature" />
    <dataitem name="Humidity" />
    <dataitem name="Action" />
  </dataitemlist>

  <channellist>
    <channel direction="TtoD" taskname="TemperatureSampler"
dataname="Temperature" interest="none:0" local="true"/>

    <channel direction="DtoT" taskname="Collector" dataname="Temperature"
interest="domain:0" local="true"/>
...
  </channellist>
</atagprogram>
```

Figure 6.3: XML declaration for the program in Figure 6.2.

6.1.2 Customizable Code Auto-generator

As stated previously in Section 3, an ATaG program consists of two parts - the *task graph* representing the properties and inter-relationships of the abstract tasks and data items, and *imperative code* expressing the details of each task and data item. This component takes the XML file generated by the GME pattern processor and generates the following files:

```
/********************/
/** Collector.java */
/********************/
package atag.apps.mainApp;

import atag.runtime.DataItem;
import atag.runtime.DataPool;
import atag.runtime.NodeInfo;

public class Collector implements Runnable {

        private DataPool m_dataPool;

        private NodeInfo m_myState;

        /** Creates a new instance of Collector */
        public Collector(DataPool dp, NodeInfo myconfig) {
                m_dataPool = dp;
                m_myState = myconfig;
        }

        public void run() {
                /*Checking for each DataItem, to see which one was produced*/
                DataItem t_dataItem;
                if((t_dataItem = m_dataPool.getData(IDConstants.T_COLLECTOR,
                        IDConstants.D_TEMPERATURE)) != null){
                        Temperature recvdTemperature =
                            (Temperature) t_dataItem.core();
                        this.handleTemperatureProduced(recvdTemperature);
                }else if((t_dataItem =
                          m_dataPool.getData(IDConstants.T_COLLECTOR,
                        IDConstants.D_HUMIDITY)) != null){
                        Humidity recvdHumidity = (Humidity) t_dataItem.core();
                        this.handleHumidityProduced(recvdHumidity);
                }
        }

        private void handleTemperatureProduced(Temperature r_Temperature) {
       /* Sample code - please write your own */
       /* You can also use the putData Code Samples below */
                int var1 = r_Temperature.getvar1();
                m_myState.debugPrint("[Collector]Got a Var1 " + var1);
        }
        private void handleHumidityProduced(Humidity r_Humidity) {
       /* Sample code - please write your own */
       /* You can also use the putData Code Samples below */
                int var1 = r_Humidity.getvar1();
                m_myState.debugPrint("[Collector]Got a Var1 " + var1);
        }
    /*********************/
    /* Sample code to be used for producing data items
       Copy-paste and uncomment to use*/
    /*********************/

    /* Sample code for producing dataItem Action
                Action m_Action = new Action();
                m_Action.setvar1(this.m_myState.getMyId());
                DataItem m_dataitem = new DataItem(IDConstants.D_ACTION,
                                IDConstants.T_COLLECTOR, m_Action);
                m_dataPool.putData(m_dataitem);
    */
}
```

Figure 6.4: Auto-generated code for the *Collector* task

- **IDConstants.java**: This contains the declarations mapping each task and data item's ID to a static variable to enhance readability.

- **[DataName].java**: For each data item, *Srijan* creates a file with the needed Java headers, and methods for serialization and deserialization of the data item. The file also contains dummy code for the contents of the data item itself, with guidelines on how to edit it.

- **[TaskName].java**: For each abstract task, *Srijan* creates a file which describes the task as a Java class. For periodic tasks, the code for ensuring that the task fires with the desired frequency is pre-built into the auto-generated code. The user only needs to edit the `handleExpiryOfTimer()` method. For tasks that are fired due to the presence of data item(s), the auto-generated code contains sample code for methods for handling the data item produced.

  In both cases, this component generates sample code for calls to the `putData()` method of the runtime system, using which the developers can make the tasks interact with the DataPool.

- **PreBuiltAtagManager.java** This file contains the code for determining the logical scope corresponding to each data item, so that the runtime can deliver it accordingly as it is produced. Note that this file does not need any editing from the application developer.

The code generated at this stage by *Srijan* is compatible with the widely used NetBeans^{TM}Java IDE, and can be easily accessed in it for editing and syntax check. Figure 6.4 shows the file auto-generated by *Srijan* for the *Collector* task of the HVAC

```
0 0014:4F01:0000:0B55 40 100 room:0 attachedSensors:TemperatureSensor
|          |          |    |    |                     |
|  Physical Address   |    |   Partition Info    Hardware Attributes
Node ID               X & Y Coordinates
```

Figure 6.5: Line describing node 0 from the system description file

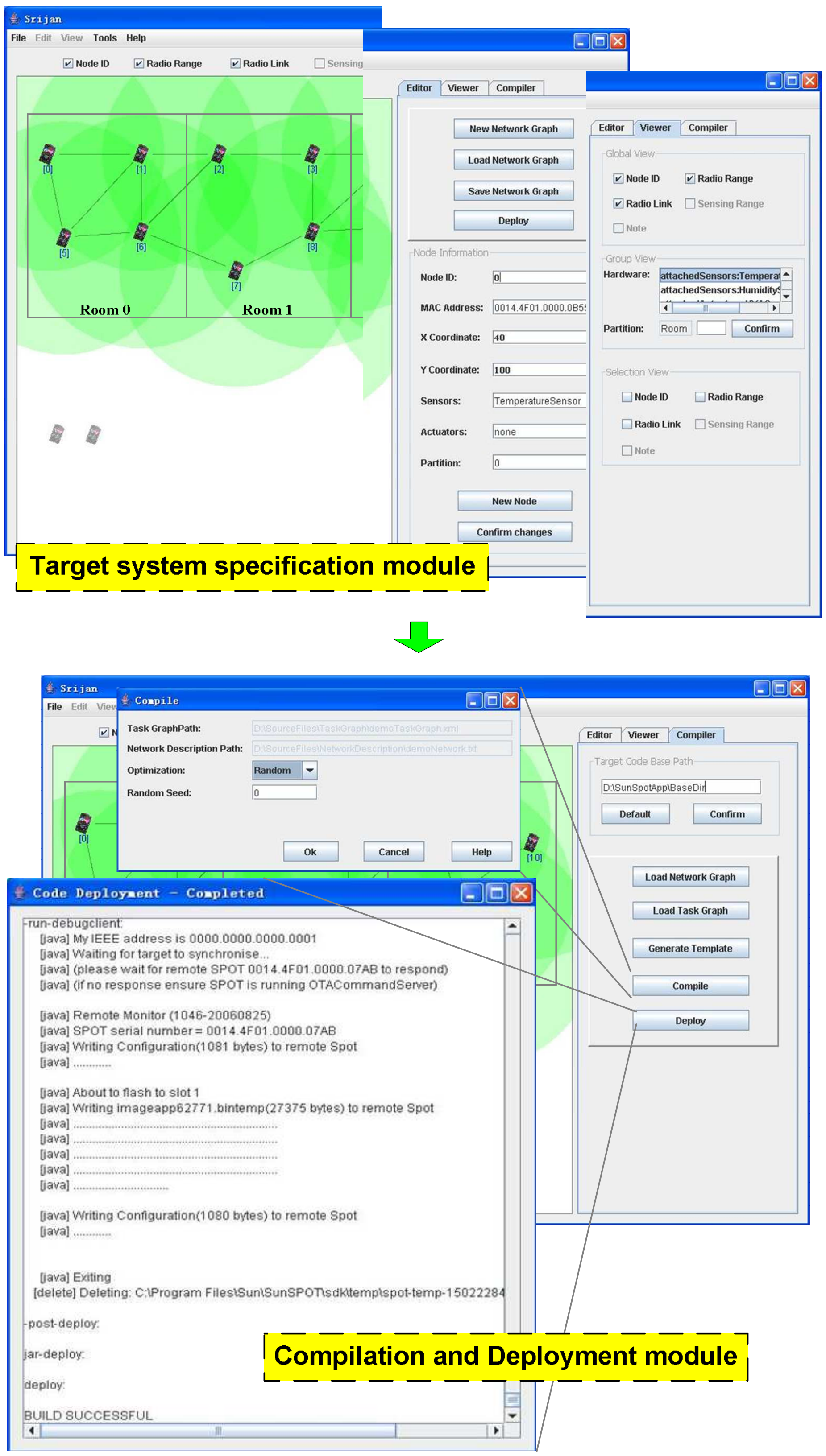
application. Note that the developer only has to insert code in the `handleHumidity-Produced()` and `handleTemperatureProduced()` methods. *Srijan* also provides code samples to copy-paste for generating the *Action* data item produced by the *Collector*.

6.1.3 Target System Description GUI

Although WSN applications are developed for a specific *purpose* (e.g. HVAC management), users of ATaG can use the same ATaG programs for a variety of *target deployments* (buildings). The ATaG compiler takes the target system description as input while allocating tasks, and performs optimizations to enhance desired metrics like system lifetime. This component of *Srijan* enables the application developer to specify the structure of the target system in a graphical manner.

The description of the target system can be uploaded into *Srijan* in the form of a plain text file. The first line of the file describes the network's global properties and each succeeding line describes the attributes of individual nodes. Currently, the global network information that can be specified includes the total number of nodes, (rectangular) area covered and (common) radio ranges. Supported node attributes include the node ID, physical ID (MAC address), physical position (coordinates), partition ID and attached hardware such as sensors and actuators. Figure 6.5 shows the format of one such line from a network description file.

The top part of Figure 6.6 shows the components of our toolkit used for specifying the target system structure. The GUI consists of two main components, a display



Node Information
Node ID: 0
MAC Address: 0014.4F01.0000.0B5
X Coordinate: 40
Y Coordinate: 100
Sensors: TemperatureSensor
Actuators: none
Partition: 0

Hardware: attachedSensors:Temperat
attachedSensors:Humidity
Partition: Room

Target Code Base Path: D:\SunSpotApp\BaseDir

Compile
Task GraphPath: D:\SourceFiles\TaskGraph\demoTaskGraph.xml
Network Description Path: D:\SourceFiles\NetworkDescription\demoNetwork.txt
Optimization: Random
Random Seed: 0

Code Deployment – Completed
```
-run-debugclient:
    [java] My IEEE address is 0000.0000.0000.0001
    [java] Waiting for target to synchronise...
    [java] (please wait for remote SPOT 0014.4F01.0000.07AB to respond)
    [java] (if no response ensure SPOT is running OTACommandServer)

    [java] Remote Monitor (1046-20060825)
    [java] SPOT serial number = 0014.4F01.0000.07AB
    [java] Writing Configuration(1081 bytes) to remote Spot
    [java] ...........

    [java] About to flash to slot 1
    [java] Writing imageapp62771.bintemp(27375 bytes) to remote Spot
    [java] ..........................................................
    [java] ..........................................................
    [java] ..........................................................
    [java] ..........................................................
    [java] ...........................

    [java] Writing Configuration(1080 bytes) to remote Spot
    [java] ...........


    [java] Exiting
  [delete] Deleting: C:\Program Files\Sun\SunSPOT\sdk\temp\spot-temp-15022284

-post-deploy:

jar-deploy:

deploy:

BUILD SUCCESSFUL
```

Figure 6.6: Network description, compilation and deployment using *Srijan*

screen at the left side of the main window and a tabbed control panel which contains three sub-panels - *Editor*, *Viewer*, and *Compiler*.

In the *Editor* panel of the GUI, there are buttons enabling the creation, uploading and saving of a network description file. The network defined by the description file is shown in the display screen. Further, the developer can add new nodes or edit a selected node. Unconfigured new nodes are shown dimmed. The selected node is highlighted and its attributes are listed in the node information area. Node attributes can be changed by editing values in corresponding text fields. Node locations can be also adjusted by simply dragging and dropping that node in the screen directly. To assist network configuration, the viewer panel provides a rich set of options to display attributes associated with nodes such as radio range, radio links between neighbors etc. These options can be combined to specify complex constraints to display information that developers might be interested in. For example, it can be configured to show radio links in a specific partition/group or between nodes with a specific type of sensor etc. Generally, the editor and viewer panel aim to provide facilities to give the application developer an intuitive view of a system's profile while configuring it.

6.1.4 Compilation and Deployment Module

The lower half of Figure 6.6 depicts the module of our toolkit that provides the application developer the ability to tune the compilation parameters and deploy the generated code to the nodes in the target system. The developer can specify the root directory of the target code using the *Base Directory* textfield at the top of the compiler panel. The *Generate Template* button below is used to generate the Java imperative code templates. The *Compile* button brings up the the compiler configuration dialog shown in the figure. The current ATaG compiler supports only the *random* option for optimization and the developer can configure the randomization seed in the textfield. *Srijan* can easily support more optimizations as they are developed. After setting the compilation parameters, the developer can click the *OK* button to initiate compilation. The integrated ATaG compiler then parses the program files and generates Java2ME code customized for each node. The generated code of each node is placed in a directory named by the node's logical ID in the root directory specified before.

For deploying the code on the Sun SPOTs, our toolkit uses a Sun SPOT basestation node for uploading it to the nodes via an OTA(over the air) command in the format `ant -DremoteId = nnnn.nnnn.nnnn.nnnn deploy` where `nnnn.nnnn.nnnn.nnnn` is the IEEE extended MAC address for Sun SPOTs. As we have associated the logical ID of the nodes with their MAC addresses in the network description file, when the *Deploy* button is clicked, the composition GUI calls the `ant deploy` command iteratively to upload node executables to corresponding nodes in the system.

6.2 Evaluation

To evaluate the performance of *Srijan*, we developed both the applications discussed in Section 3.3 on it. For each of the applications, we performed the complete end-to-end development – starting from specifying the ATaG task graph to deployment of code on the nodes – using *Srijan*. We used a Pentium-4 2.8 GHz laptop with 1GB of RAM running Windows XP for our evaluation. The deployment was done onto the Sun SPOT [93] nodes, with a 180 MHz 32 bit ARM920T processor, 512K RAM and 4M Flash memory. The nodes run the Squawk Java virtual machine directly out of flash memory, and can run programs written using J2ME libraries. The Sun SPOT base station was used to deploy the code over-the-air (OTA) to the SPOTs. We used the Java hProf profiler [47] for measuring execution time.

During our experiments, we collected a variety of statistics. The first metric was the time taken by the toolkit to **a)** create the auto-generated imperative code code templates, **b)** allocate tasks to the nodes and generate per-node customized Java files, and **c)** generate the Java bytecode for each node and deploy it over the air. In addition to the above times, we also collected statistics regarding the amount of total code that was written by the application developer versus the code auto-generated by *Srijan*. Although the *line-of-code* metric is more a measure of the power of the ATaG compiler, we report the numbers because **a)** these numbers are of our J2ME-targeted implementation of the ATaG compilation framework, and **b)** this emphasizes the power of the ATaG macroprogramming paradigm which is made accessible to the application developer in a graphical manner by our toolkit.

In addition to the above objective metrics, we also measured the time it took for an application developer using *Srijan* to specify the ATaG task graph as well as the time taken in customizing the imperative code generated by it. We acknowledge that these timings are variable from person to person, and intend to obtain more such data following the public release of our software to get a better idea of the burden to the programmer when using our toolkit.

The data from our experiments is summarized in Table 6.1. Note that the time taken by *Srijan* to generate the files are within acceptable limits, and are limited only by the hardware it is being run on, and in the case of deployment, also on the Java compiler used by the Sun SPOT SDK. More importantly, the developer had to write only a very small fraction of Java source files. The *total code* deployed on each node consists of three components: **a)** Base Template Code — containing the DART libraries,

	HVAC	**Traffic**
Imperative Code Gen. Time (ms)	1766	3422
Node-Specific Code Gen. Time (ms)	31967	77089
Per-node Deployment Time (s)	21	23
Source Files Edited by Developer	11	18
Total Number of Source Files	57	64
Lines of Application-specific Auto-generated Code	569	1019
Lines of Application-specific Code Written by Developer	60	81
Total Lines of Code	3433	3904
Task Graph Specification Time (min)	10	25
Imperative Code Editing Time (min)	17	60

Table 6.1: Costs involved in various stages of application development using *Srijan*

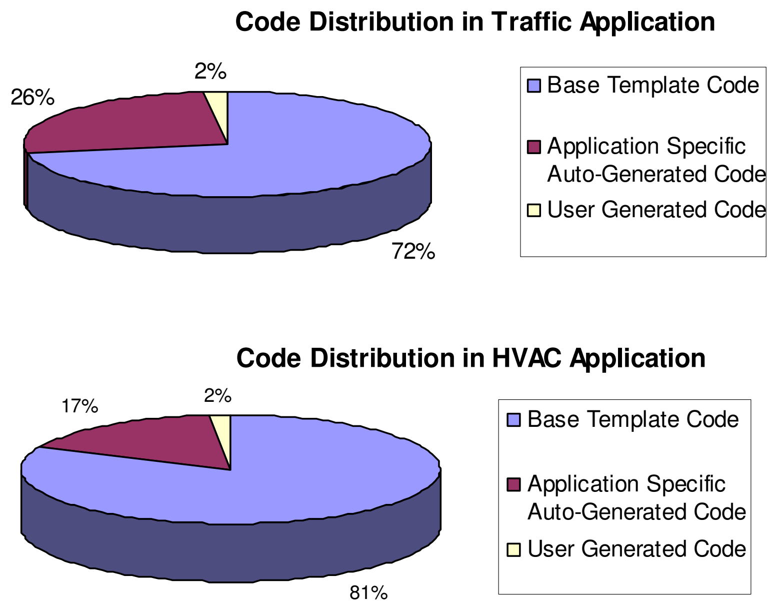


Figure 6.7: Distribution of coding effort

b) Application-specific Auto-generated Code — generated by *Srijan*, and **c)** User-generated Code — written by the application developer to specify the details of the task and data. Figure 6.7 shows that the user-generated code is only around 2% of the total code. Even if we neglect the library code, *Srijan* generated $> 90\%$ of the application-specific code in each case. The importance of the time taken by the application developer in specifying the task graph and customizing the auto-generated code is highlighted by the fact that under normal circumstances, *Srijan* will be used by domain experts, e.g. civil engineers, who would have taken much more time customizing the runtime protocols and figuring out the task placements if it was not available as part of *Srijan*. We believe that our experiments demonstrate that our toolkit makes application development for WSNs more convenient for the domain expert, and we look forward to feedback from developers who download and use our toolkit.

Chapter 7

Concluding Remarks

WSN macroprogramming holds the promise of getting application development on sensor networks in the reach of domain experts who may not necessarily be experts in distributed computing, However 'the ease-of-use provided by macroprogramming comes at a cost in terms of the effort needed to convert the developer's specifications to the intended system. The work in this thesis attempts to identify the issues raised in this context and address them. Focusing on data-driven macroprogramming, using which a wide variety of sense and-respond applications can be specified, we have proposed a compilation framework that clearly demarcates the various steps involved in the process. Focusing on the crucial task-mapping phase of compilation, we have provided mathematical formulations for solving the problem optimally using our modeling technique. To compute the task mappings in less time, we have also presented heuristics, both for the single-path and multipath cases Finally, we have incorporated our work on macroprogram compilation into a graphical toolkit for WSN application development.

The work in this thesis is the first step in a long journey. Future work on this topic can be in the following related areas:

- **Supporting Advanced Primitives for Data-driven Macroprogramming:** The current set of ATaG primitives supported by our compilation framework, although powerful, can be extended upon. As a specific case, the logical scopes currently supported are *static* (e.g., all nodes in the 3rd floor with temperature sensors). However, allowing the application developer to use dynamic scopes (e.g., all nodes in the 3rd floor with a temperature value > 20 degrees) will help them specify applications more easily. The issues involved in translating these annotations are worth exploring.

- **Supporting Task Migration:** The task-mapping techniques proposed by us assume that this initial *good* task-mapping will help the WSN stay alive for a long time. However, since sensor networks operate in a dynamic environment, it is imperative that the DART runtime be extended to support the migration of tasks based on the energy situation of the networked sensor system. This will involve addressing the questions of *when* to migrate tasks, *which* tasks to migrate, and *how* to migrate them; all while making sure that the semantics of the original macroprogram are preserved. The work in this area can also lead to the development of *distributed task mapping algorithms*, with clear bounds on the tradeoff between the quality of the solution (e.g., revised system lifetime) and the range from which a node has to obtain information due to the absense of a global view of the system.

- **Supporting Compilation of Truly Distributed Primitives:** In our work, we have assumed that the processing of the data in the WSN occurs in discrete locations (e.g., in a simple temperature monitoring application, a *Collector* runs on exactly one node in each room, and computes the average of the readings received from other nodes in the room). However, we believe that the power of the networked sensor system can be utilized to a greater extent if the developer is able to specify his application using *distributed* tasks (e.g., "*The nodes in the room collaboratively compute the average temperature*"). Note that in this case, the *Collector* still produces the same output as before (average temperature in the room), but the compiler is now free to break the process of average-computation into smaller tasks, which can be distributed among all nodes in the room for energy-efficiency. While such abstractions make the programming language more powerful, they also bring with them added challenges for the compiler designer. For example, how will the compiler know which way to break such *distributed* tasks? What are the best techniques of placing them?

The future of networked sensing is a promising one, with aims including safer buildings, better flowing highways, and cleaner lakes for all; and the macroprogram compiler will play a large part in bringing the power of the WSN to the masses.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, and A. Wood. EnviroTrack: An Environmental Programming Model for Tracking Applications in Distributed Sensor Networks. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[2] Zoe Abrams and Jie Liu. Greedy is good: On service tree placement for in-network stream processing. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 72, Washington, DC, USA, 2006. IEEE Computer Society.

[3] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: Research challenges. *Ad Hoc Networks Journal*, 2(4):351–367, October 2004.

[4] A. Alessandri, A. di Febbraro, A. Ferrara, and E. Punta. Nonlinear optimization for freeway control using variable-speedsignaling. *IEEE Transact. on Vehicular Technology*, 48(6), Nov 1999.

[5] John-Paul Arp and Bradford G. Nickerson. A user friendly toolkit for building robust environmental sensor networks. In *CNSR '07: Proceedings of the Fifth Annual Conference on Communication Networks and Services Research*, pages 76–84, 2007.

[6] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner. The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems. In *Workshop on End-to-end Sense-and-respond Systems (EESR)*, 2005.

[7] Amol Bakshi, Animesh Pathak, and Viktor K. Prasanna. System-level support for macroprogramming of networked sensing applications. In *Int. Conf. on Pervasive Systems and Computing (PSC)*, 2005.

[8] Amol Bakshi and Viktor K. Prasanna. *Architecture-Independent Programming for Wireless Sensor Networks*. Wiley Series on Parallel and Distributed Computing. Wiley, 2008.

[9] R. Barr, Z. J. Haas, and R. van Renesse. Jist: an efficient approach to simulation using virtual machines. *Softw. Pract. Exper.*, 35(6), 2005.

[10] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using java byte code. In *Proc. of the* 12^{nd} *Euromicro Conf. on Real-Time Systems*, 2000.

[11] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, March 1981.

[12] Eric Bouillet, Mark Feblowitz, Zhen Liu, Anand Ranganathan, Anton Riabov, and Fan Ye. A semantics-based middleware for utilizing heterogeneous sensor networks. In *Proc. of the* 3^{rd} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2007.

[13] Azzedine Boukerche. *Algorithms and Protocols for Wireless and Mobile Systems*. Chapman & Hall/CRC 2005, 2005.

[14] Btnodes - a distributed environment for prototyping ad hoc networks. `http://www.btnode.ethz.ch`.

[15] P. Buonadonna, D. Gay, J.M. Hellerstein, W. Hong, and S. Madden. TASK: sensor network in a box. In *Second European Workshop on Wireless Sensor Networks, EWSN 2005.*, February 2005.

[16] J. Caffrey, R. Govindan, E. Johnson, B. Krishnamachari, S. Masri, G. Sukhatme, K. Chintalapudi, K. Dantu, S. Rangwala, A. Sridharan, N. Xu, and M. Zuniga. Networked sensing for structural health monitoring. In *Proceedings of the 4th International Workshop on Structural Control*, pages 57–66, June 2004.

[17] Ioannis Chatzigiannakis, George Mylonas, and Sotiris E. Nikoletseas. jWebDust : A java-based generic application environment for wireless sensor networks. In *DCOSS*, pages 376–386, 2005.

[18] Rohit Chaudhri, Yang Yu, Silviu Chiricescu, Loren J. Rittle, and Rajit Gadh. Mobile device-centric exercise monitoring with an external sensor population. In *International Conference on Distributed Computing in Sensor Systems (Poster)*, 2007.

[19] Elaine Cheong, Edward A. Lee, and Yang Zhao. Joint modeling and design of wireless networks and sensor node software. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

[20] Jatuporn Chinrungrueng, Udomporn Sunantachaikul, and Satien Triamlumlerd. Smart parking: An application of optical wireless sensor network. In *Applications and the Internet Workshops, 2007. SAINT Workshops 2007. International Symposium on*, January 2007.

[21] W. Choi, P. Shah, and S.K. Das. A framework for energy-saving data gathering using two-phase clustering in wireless sensor networks. In *Proc. of the* 1^{st} *Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services (MOBIQUITOUS)*, 2004.

[22] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Proc. of the* 22^{nd} *Int. Conf. on Software Engineering (ICSE)*, 2000.

[23] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. Tinylime: bridging mobile and sensor networks through middleware. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pages 61–72, 2005.

[24] M. Dermibas. Wireless sensor networks for monitoring of large public buildings. Technical report, University at Buffalo, 2005.

[25] A. Deshpande, C. Guestrin, and S. Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering*, 28(1), 2005.

[26] DJProf Java Profiler, `www.mcs.vuw.ac.nz/ djp/djprof/`.

[27] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *1st IEEE Workshop on Embedded Networked Sensors*, 2004.

[28] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network - a toolkit for the development of WSNs. In *European Workshop on Wireless Sensor Networks (EWSN)*, 2007.

[29] C. Efthymiou, S. Nikoletseas, and J. Rolim. Energy balanced data propagation in wireless sensor networks. *Wireless Networks (WINET) Journal, Special Issue on "Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks*, 2006.

[30] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[31] J. Elson and D. Estrin. Time synchronization in wireless sensor networks. In *International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing*, April 2001.

[32] Porting tinyos to the ens box architecture. `http://research.cens.ucla.edu/projects/2007/Systems/TinyOS/`.

[33] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *Proc. of the* 5^{th} *Int. Conf. on Mobile computing and networking (MobiCom)*, 1999.

[34] James R. Evans and Edward Minieka. *Optimization Algorithms for Networks and Graphs, Second Edition*. CRC Press, 2nd Edition, 1992.

[35] Folding@home. `http://folding.stanford.edu/`.

[36] Christian Frank and Kay Römer. Solving generic role assignment exactly. In *International Parallel Distributed Processing Symposium (IPDPS)*, 2006.

[37] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[38] G. Giorgetti, G. Manes, J.H. Lewis, Sal T. Mastroianni, and S.K.S. Gupta. The personal sensor network: A user-centric monitoring solution. In *2nd International Conference on Body Area Networks*, June 2007.

[39] The Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme.

[40] GRATIS: Graphical development environment for tinyos. http://www.isis.vanderbilt.edu/projects/nest/gratis/index.html.

[41] Habitat Monitoring on the Great Duck Island. `www.greatduckisland.net`.

[42] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *2nd ACM Conference on Embedded Networked Sensor Systems*, 2004.

[43] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *Proc. of the* 1^{st} *Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[44] Gumstix. `http://www.gumstix.com`.

[45] Tian He, Sudha Krishnamurthy, Liqian Luo, Ting Yan, Lin Gu, Radu Stoleru, Gang Zhou, Qing Cao, Pascal Vicaire, John A. Stankovic, Tarek F. Abdelzaher, Jonathan Hui, and Bruce Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[46] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[47] HPROF: A heap/cpu profiling tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html.

[48] T. T. Hsieh. Using sensor networks for highway and traffic applications. *IEEE Potentials*, 23(2), 2004.

[49] Crossbow imote2. http://www.xbow.com/Products/productdetails.aspx?sid=253.

[50] S. Sitharama Iyengar and Ricahrd R. Brooks. *Distributed Sensor Networks (Chapman & Hall/Crc Computer and Information Science)*. Chapman & Hall/CRC, 2004.

[51] Vivek Jain, Ratnabali Biswas, and Dharma P. Agrawal. Energy efficient and reliable medium access for wireless sensor networks. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2007.

[52] N. Jiang and M. Parashar. Enabling applications in sensor-based pervasive environments. In *BROADNETS 2004: Workshop on Broadband Advanced Sensor Networks (BaseNets 2004)*, October 2004.

[53] P. Kachroo and K. Ozbay. *Feedback Ramp Metering in Intelligent Transportation Systems*. Plenum Pub Corp, 2004.

[54] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous computing systems. In *Sixth Heterogeneous Computing Workshop (HCW '97)*, April 1997.

[55] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. ACM/IEEE MobiCom*, August 2000.

[56] J. Koshy and R. Pandey. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of ACM SenSys 2005, San Diego, CA*, 2005.

[57] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. *SIGPLAN Not.*, 42(6):200–210, 2007.

[58] Bhaskar Krishnamachari. *Networking Wireless Sensors*. Cambridge University Press, 2006.

[59] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[60] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin G&#252;n Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM Press.

[61] Kian Hsiang Low, W.K. Leow, and Jr. M.H. Ang. Autonomic mobile sensor network with self-coordinated task allocation and execution. *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, 36(3):315–327, May 2006.

[62] LP Solve. `http://lpsolve.sourceforge.net/`.

[63] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3), 2006.

[64] Perng-Yi Richard Ma, E.Y.S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, January 1982.

[65] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A language for data-driven sensor network programs. In *Technical Report TR-13-06, Harvard University Technical Report*, 2006.

[66] C. Manzie, H. C. Watson, S. K. Halgamuge, and K. Lim. On the potential for improving fuel economy using a traffic flow sensor network. In *Proc. of the Int. Conf. on Intelligent Sensing and Information Processing*, 2005.

[67] Dustin McIntire, Kei Ho, Bernie Yip, Amarjeet Singh, Winston Wu, and William J. Kaiser. The low power energy aware processing (leap)embedded networked sensor system. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 449–457, New York, NY, USA, 2006. ACM.

[68] uC/OS-II RTOS, http://www.ucos-ii.com/.

[69] L. Mottola and G. P. Picco. Logical Neighborhoods: A programming abstraction for wireless sensor networks. In *Proc. of the the* 2^{nd} *Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2006.

[70] L. Mottola and G. P. Picco. Programming wireless sensor networks with Logical Neighborhoods. In *Proc. of the* 1^{st} *Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense)*, 2006.

[71] Luca Mottola, Animesh Pathak, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Enabling Scoping in Sensor Network Macroprogramming. Technical report. Submitted for publication. Available at http://pgroup.usc.edu/atag, 2006.

[72] Luca Mottola, Animesh Pathak, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Enabling Scoping in Sensor Network Macroprogramming. In *Fourth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, 2007.

[73] Piyush Naik and Krishna M. Sivalingam. A survey of mac protocols for sensor networks. *Wireless sensor networks*, pages 93–107, 2004.

[74] G.L. Nemhauser, A.H.G. RinnooyKan, and M.J. Todd, editors. *Optimiztions: Handbooks in Operations Research and Management Science*, volume 1. North-Holland, 1989.

[75] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. of the* 4^{th} *Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[76] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc of the* 1^{st} *Int. Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[77] Ryan Newton, Lewis Girod, Michael Craig, Greg Morrisett, and Samuel Madden. Design and evaluation of a compiler for embedded stream programs. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, 2008.

[78] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Compiling macroprograms using the ATaG compilation framework, http://pgroup.usc.edu/atag. Technical report, University of Southern California, 2007.

[79] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Third IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS 2007)*, 2007.

[80] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Proc. of the* 3^{rd} *Int. Wkshp. on Sensor Networks and Systems for Pervasive Computing (PerSens - colocated with IEEE PERCOM)*, 2007.

[81] Animesh Pathak and Viktor K. Prasanna. Issues in Designing a Compilation Framework for Macroprogrammed Networked Sensor Systems. In *Proc. of the the* 1*st* *Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense)*, 2006.

[82] Animesh Pathak and Viktor K. Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2008.

[83] Olivier Powell, Pierre Leone, and José D. P. Rolim. Energy optimal data propagation in wireless sensor networks. *J. Parallel Distrib. Comput.*, 67(3):302–317, 2007.

[84] Sushil K. Prasad and Akshaye Dhawan. Distributed algorithms for lifetime of wireless sensor networks based on dependency structure among cover sets. In *Intl High Performance Computing (HiPC)*, 2007.

[85] M. Rahimi, M. Hansen, W.J. Kaiser, G.S. Sukhatme, and D. Estrin. Adaptive sampling for environmental field estimation using robotic sensors. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, August 2005.

[86] C.P. Ravikumar and A.K. Gupta. Genetic algorithm for mapping tasks onto a reconfigurable parallel processor. *IEE Proceedings on Computers and Digital Techniques*, 142(2):81–86, March 1995.

[87] IST EU RUNES Project. www.ist-runes.org.

[88] M.Y. Aal Salem, D. Everitt, and A.Y. Zomaya. An interoperability framework for sensor and umts networks. In 3^{rd} *International Conference on Wireless and Mobile Communications*, 2007.

[89] A. Savvides, C.-C. Han, and M. B. Srivastava. Dynamic fine-grain localization in ad-hoc networks of sensors. In *Proc. 7th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 2001.

[90] Seti@home. `http://setiathome.berkeley.edu/`.

[91] Doug Simon and Cristina Cifuentes. The squawk virtual machine: Java$^{\text{TM}}$ on the bare metal. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–151, New York, NY, USA, 2005. ACM Press.

[92] Crossbox StarGate, http://www.xbow.com/Products/XScale.html.

[93] Sun$^{\text{TM}}$Small Programmable Object Technology (Sun SPOT), `www.sunspotworld.com`.

[94] Benjamin Khoo Boon Tat, Bharadwaj Veeravalli, Terence Hung, and Simon See Chong Wee. A co-ordinate based resource allocation strategy for grid environments. In 6th *IEEE International Symposium on Cluster Computing and Grid (CCGRID)*, pages 561–567, May 2006.

[95] Crossbow telosb mote. `http://www.xbow.com/Products/productdetails.aspx?sid=252`.

[96] Yuan Tian, E. Ekici, and F Ozguner. Energy-constrained task mapping and scheduling in wireless sensor networks. In *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference RPMSN Workshop*, 2005.

[97] Tenet: An Architecture for Tiered Embedded Networks, http://tenet.usc.edu.

[98] Torben Weis, Mirko Knoll, Andreas Ulbrich, Gero Mhl, and Alexander Brndle. Rapid prototyping for pervasive applications. *IEEE Pervasive Computing*, 6(2):76–84, 2007.

[99] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. of the* 2nd *Int. Conf. on Mobile systems, applications, and services (MOBISYS)*, 2004.

[100] Kamin Whitehouse, Feng Zhao, , and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *European Workshop on Wireless Sensor Networks (EWSN)*, 2006.

[101] Alec Woo, Siddharth Seth, Tim Olson, Jie Liu, and Feng Zhao. A spreadsheet approach to programming and managing sensor networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, New York, NY, USA, 2006.

[102] Yang Yu and Viktor K. Prasanna. Energy-balanced task allocation for collaborative processing in wireless sensor networks. *MONET*, 10(1-2):115–131, 2005.