HAL
open science

# High-Level Application Development for Sensor Networks: Data-Driven Approach

Animesh Pathak, Viktor K. Prasanna

## To cite this version:

Animesh Pathak, Viktor K. Prasanna. High-Level Application Development for Sensor Networks: Data-Driven Approach. Nikoletseas, Sotiris and Rolim, José D.P. Theoretical Aspects of Distributed Computing in Sensor Networks, Springer Berlin Heidelberg, pp.865-891, 2011, Monographs in Theoretical Computer Science. An EATCS Series, 978-3-642-14848-4. 10.1007/978-3-642-14849-1_26 . hal-00723799

**HAL Id: hal-00723799**

**https://inria.hal.science/hal-00723799**

Submitted on 10 Dec 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# High-level Application Development for Sensor Networks: Data-driven Approach

Animesh Pathak and Viktor K. Prasanna

**Abstract** Owing to the large scale of networked sensor systems, ease of programming remains a hurdle in their wide acceptance. High-level application development techniques, or *macroprogramming* provides an easy to use high-level representation to the application developer, who can focus on specifying the behavior of the *system*, as opposed to the *constituent nodes* of the wireless sensor network (WSN).

This chapter provides an overview of the current approaches to high-level application design for WSNs, going into the details related to data-driven macroprogramming. Details of one such language are provided, in addition to the approach taken to the compilation of data-driven macroprograms to node-level code. An implementation of the modular compilation framework is also discussed, as well as a graphical toolkit built around it that supports data-driven macroprogramming. Through experiments, it is shown that the code generated by the compiler matches hand-generated implementations of the applications, while drastically reducing the time and effort involved in developing real-world WSN applications.

## 1 Introduction

Wireless sensor networks (WSNs) enable low cost, dense monitoring of the physical environment through collaborative computation and communication in a network of autonomous sensor nodes, and are an area of active research [7]. Owing to the work done on system-level services such as energy-efficient medium access [25] and data-propagation [45] techniques, sensor networks are being deployed in the real world, with an accompanied increase in network sizes, amount of data handled, and the

Animesh Pathak
INRIA Paris-Rocquencourt, France, e-mail: animesh.pathak@inria.fr

Viktor K. Prasanna
University of Southern California, USA, e-mail: prasanna@usc.edu

variety of applications [28, 46, 19, 26]. 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 deployed in a building or a highway. Throughout this chapter, the term *domain expert* will be used 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.

## *1.1 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 [16], galsC or Java to write the program, directly interacting with the node-level services stated earlier, or a middleware [50, 14, 36] 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. This is believed to be 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.

One of the earliest toolkits proposed to reduce the programming effort was the Sensor Network Application Construction Kit (SNACK) [20], 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 com-

ponents, 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 [33] 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. In addition to the above, some graphical toolkits have also been proposed for WSN application development. The authors of Viptos [11] allow developers to model and simulate TinyOS applications in a graphical manner. A similar functionality is provided by GRATIS [18] where developers can use GME for easy modeling of TinyOS applications.

## *1.2 High-Level Abstractions for WSNs*

In spite of the tools available for easing node-level application development, the application developer is still 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. Several techniques have been proposed to provide a higher-level view of the sensor network. TinyDB [32] and Cougar [54] were the first works to abstract the sensor field as a database, allowing the user to execute queries over the sensed data in an SQL-like manner. The Task [9] 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. SenQ [53] enables user-driven and peer-to-peer in-network query issue by wearable interfaces and other resource-constrained devices. Complex virtual sensors and user-created streams can be dynamically discovered and shared, and SenQ is extensible to new sensors and processing algorithms.

Semantic streams [51] presents each user with a 3-D rendering of the sensors in the testbed as well as all predicates that are queryable. The work in [52] builds on it by providing a spreadsheet approach to programming and managing data-querying applications in WSNs. In semantic middleware [6], 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 [10] 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.

In addition to database-like abstractions, another active area of resarch for WSNs has been abstractions for the *target tracking* applications. EnviroSuite [30] 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 el-

ements they represent. The EnviroSuite Compiler (EIPLC) 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. The recent EnviroMic [31] application focuses on a distributed acoustic monitoring, storage, and trace retrieval system designed for disconnected operation.

Moving beyond domain-specific applicaion of system-level thinking, there have been attempts to provide more general-purpose high-level appliction design for sensor networks. The work on SensorWare [8] and State-centric programming [29] provided some of the initial thoughts on the matter. This was followed by increased research in the field of sensor network *macroprogramming*, which 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.

*Kairos* [21] (and later, Pleiades [27]) is an imperative, control driven macroprogramming 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.

*Regiment* [41] 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 [39], the authors introduced the TML intermediate language to represent the actions being performed at individual nodes. Regiment programs can be seen as *data flow graphs*, with primitives such as **afold** combining functions and data on actual nodes to produce data. The work in [40] extends this to the *WaveScript* language which addresses applications working on live data streams.

In their work on COSMOS [1], the authors have presented the *mPL* macroprogramming language and the *mOS* operating system which can be used to program WSNs by way of task graphs, as long as all nodes of a single type have the same set of tasks running on them. Finally, MacroLab [22] (now supported by MacroDebugging [47]) provides a Matlab-like interface to WSN application developers, so that they can use operations such as `addition`, `max`, and `find` on sensor data addressed as *macrovectors*. This paradigm focuses on accessing and operating on data presented in matrix form, sometimes using different implementations (centralized versus distributed) of the same operation (e.g. `max`). The work in ATaG [3] is focused on *data-driven macroprogramming* (discussed in detail in Section 2), 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.

For a detailed discussion of the various techniques available for application development on sensor networks, the reader is recommended to read the survey by Mottola *et. al* [38].

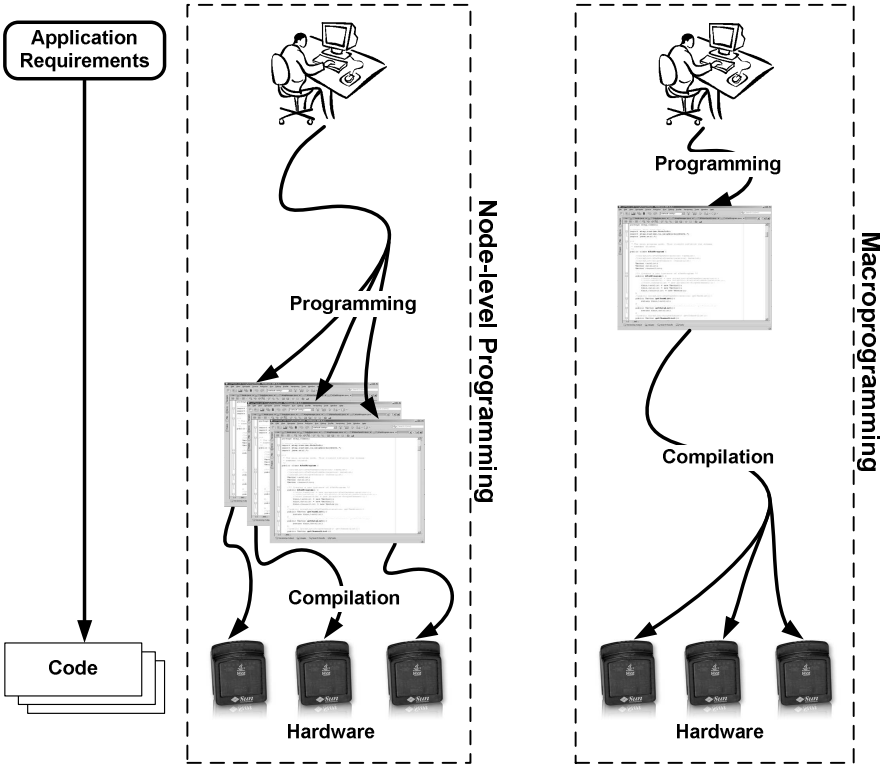



**Fig. 1** Comparing node-centric and macro- programming.

## *1.3 Macroprogram Compilation*

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*. The macroprogram compilation process has several challenges (highlighted in [43]), which must be addressed by the designers of compilation frameworks for macro-programming 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 Fig. 1, 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.

Data-driven macroprogramming languages allow 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. The focus of the rest of this chapter is to discuss in detail the design, implementation and evaluation of a compilation framework to support a data-driven macroprogramming model called the *Abstract Task Graph* (ATaG) [3], whose salient features are described in Section 2. Overall, this chapter discusses the following:

- A general framework for compilation data-driven macroprogramming languages like ATaG. An overview of the compilation process is given in Section 3. The framework breaks down the process of converting the high-level specification to node-level functionality into a set of independent, isolated 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. The compilation framework is described in detail in Section 4.
- A demonstration of the flexibility and generality of the above framework by describing an end-to-end solution for compiling ATaG macroprograms. The proof-of-concept compiler, obtained by instantiating the different modules in the 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.
- Section 5 presents the design and implementation of *Srijan* – a graphical toolkit for WSN application development.
- Section 6 shows results from developing two realistic applications – building environment management (HVAC) [15] and highway traffic management [24]. The functionality of the compiler is assessed by inspecting and comparing the auto-generated code against a manually developed version of the same. The 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).

The details of the data-driven programming model of ATaG are discussed next.

## 2 Data-driven Macroprogramming

As discussed in the previous section, macroprogramming of WSNs is an active area of research, with several programming paradigms currently being investigated. In this chapter, we focus on the *data-driven macroprogramming* paradigm, where the

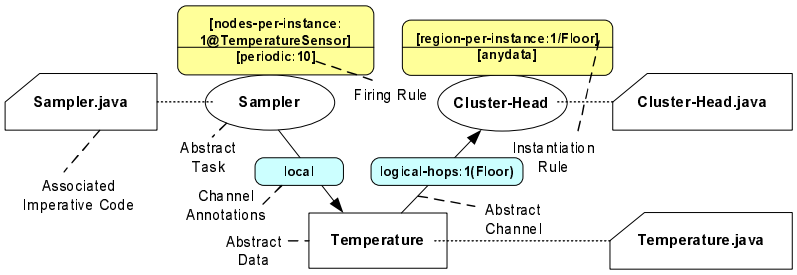



**Fig. 2** ATaG program for data-gathering

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. This technique is shown to be especially useful in specifying a wide range of sense-and-respond applications [42]. The specific data-driven macroprogramming that we focus on here is called the *Abstract Task Graph* (ATaG) [3]. 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) [2]. The compilation of 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.

## *2.1 Programming Model*

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 Fig. 2 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 [35], 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.

Fig. 2 illustrates an example ATaG program specifying a data gathering application [12] 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 the 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 the 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.

## *2.2 Runtime System*

The node-level code output by the ATaG compiler is designed to run atop a supporting runtime hiding the underlying, platform-specific details. Fig. 3 depicts the

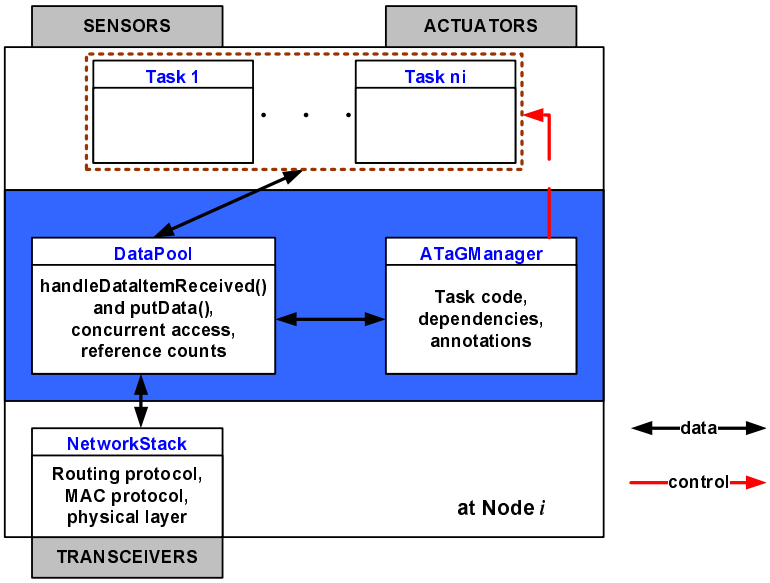



**Fig. 3** DART: Data-driven ATaG run-time system.

architecture of the Data-driven ATaG Runtime (DART) [2]. 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* [37, 36] 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.

## 3 Compilation Process

In the previous section, we described the ATaG data-driven macroprogramming paradigm. In this section, we provide a formal definition of the process of compiling data-driven macroprograms to node-level code using the application given in Fig. 2 as example.

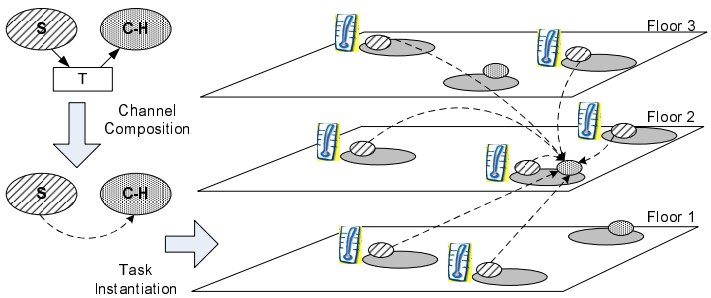



**Fig. 4** An example illustrating the compilation process of our sample program.

### *3.1 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.

### *3.2 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 Assignments:** 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.

### *3.3 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 this approach to the data-driven macroprogram compilation problem, the following major steps are envisaged.

**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, each *path* $AT_i \rightarrow AD_k \rightarrow AT_j$ in the abstract task graph is converted 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.

The *instantiated task graph* (ITaG) is 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 enables the use of modified versions of classical techniques meant for analyzing task graphs.

For example, for the application in Fig. 4, since there are seven nodes with attached temperature sensors, $f(AT_1,N) = 7$, following the **`1@TemperatureSensor`** 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 $IC$ (it shows 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. Fig. 4 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.

### 3.4 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.

The following section describes how the components of the compilation framework act together to produce the outputs from the inputs, using the ITaG notation internally, and the implementation details of the ATaG compiler.

## 4 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, first the different steps involved in the compilation of ATaG programs were identified by factoring out orthogonal concerns and mechanisms. Next, considering the decomposition obtained, a modular compilation framework was designed, upon which the construction of the ATaG compiler was based. In this section, we describe the different modules of the framework (illustrated in Fig. 5), based on the problem definition of Section 3. The compilation stages are encapsulated in separated modules, and defined generic interfaces between them so as to minimize inter-module dependencies. The modules are as follows:

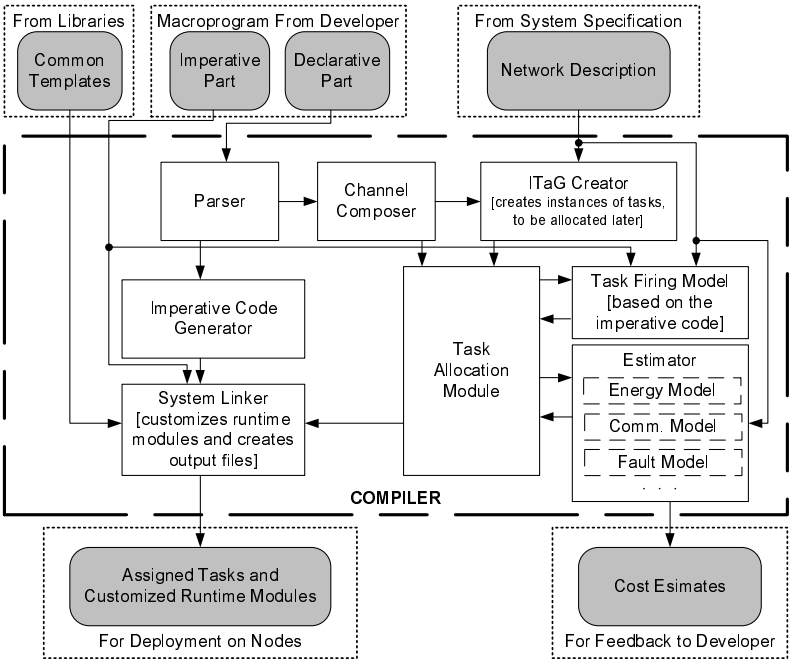



**Fig. 5** The ATaG Compilation Framework.

**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 the current implementation, the declarative part of the ATaG program is specified using XML. This allows an easy integration of tools for the automated generation of XML specifications from graphical representations. The 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 the prototype implementation, the imperative part of an ATaG program is expressed using Java. As such, the 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 3.

Depending on the actual channel annotations supported, the 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.

The 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 the 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 this 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 the 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 the framework, more sophisticated mechanisms can be plugged in to achieve performance goals specified by the application designer. Several task-mapping algorithms for improved performance have also been proposed [44].

**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. The 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 the prototype system, both ends of the spectrum were implemented. On one hand, there is 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, there is also a wrapper around SWANS/Jist [4]: a simulator able to run unmodified Java code on top of a simulated network. This plug-and-play capability highlights the power of the framework.

**Task Firing Model.** It would appear that if one knew the exact paths taken by the data items, one 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 [13, 5].

**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 the current 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.

## 5 *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 macroprogram-

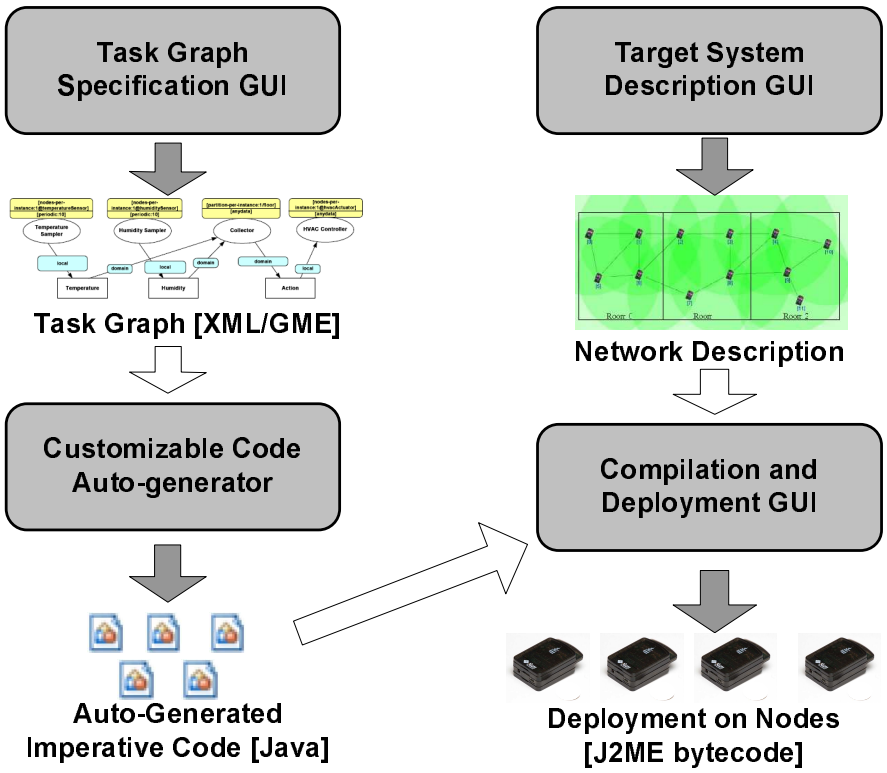



**Fig. 6** Overview of application development using *Srijan*

ming 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 the macroprogram compilation framework discussed above has been incorporated into *Srijan* (named after the Sanskrit word for creation), an easy-to-use graphical front-end to the various steps involved in developing an application using ATaG. Fig. 6 shows the various components of this toolkit. 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 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*, the Generic Modeling Environment (GME) [17] has been customized so that developers can easily specify their abstract task graphs, complete with channel and task annotations. Fig. 7 shows how a developer can specify the task graph of a building environment management application (detailed in Section 6.1) using the Srijan GUI. Once the details of the task graph are specified, *Srijan* generates an equivalent XML file, which can be used by the other modules.

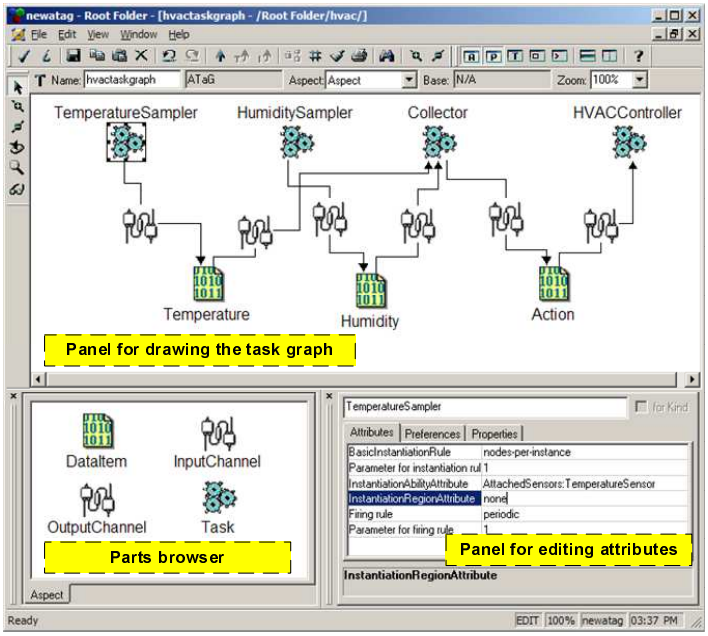



**Fig. 7** Building Environment Management (HVAC) application in the Srijan task description GUI

**Network Description, Compilation, and Deployment GUI.** The second part of *Srijan*(shown in Fig. 8) allows the developer to perform a set of actions. First, he can graphically specify the target network description, including the attributes of each node (alternatively, he can upload the specifications in a file). Secondly, the toolkit uses the task graph to generate a separate Java file for each task and data item with auto-generated communication and task-firing code.

Using the GUI, the developer can then invoke the compiler with the necessary parameters (optimization option, randomizer seed, etc.), which results in the generation of a set of Java files, one for each node in the target system, including the task and data code, as well as the customized runtime system modules. Finally, the developer can use *Srijan* to generate the bytecode for each node and deploy it to each Sun SPOT [49] over the air. The toolkit is currently under actively development, and has been released for download [48].

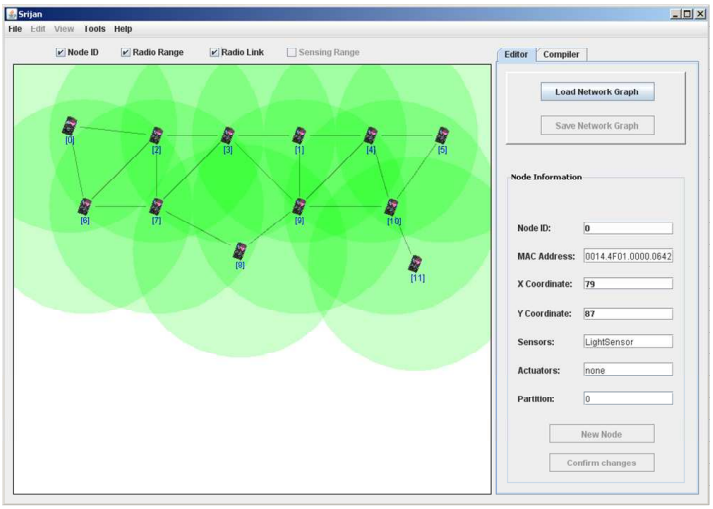



**Fig. 8** Network description, compilation and deployment using *Srijan*

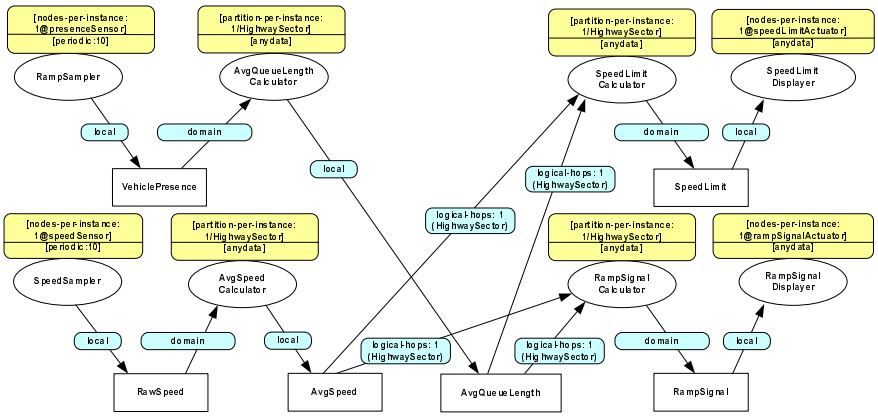



**Fig. 9** An ATaG program for highway traffic management.

# 6 Evaluation

## *6.1 Reference Applications*

To demonstrate the effectiveness of the ATaG compiler, we consider two non-trivial applications, and report on the *functionality* of the code generated, as well as the *performance* of the compilation process.

The first application, illustrated in Fig. 9, describes a highway traffic management system. In this case, two different sub-goals must be achieved - regulating the speed of vehicles on the highway by controlling speed limit displays, and control-

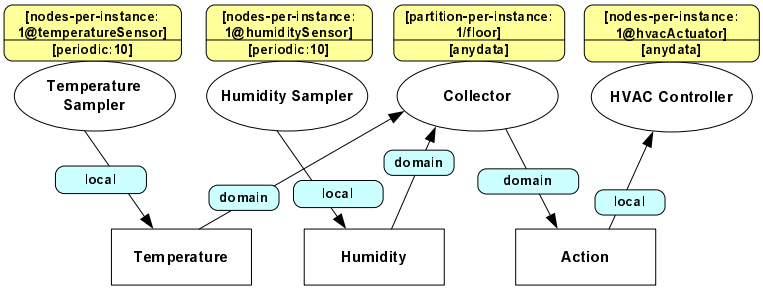



**Fig. 10** An ATaG program for building environment management.

ling 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 *AvgQueueLengthCalculator* tasks), and then delivered to tasks deciding the actions taken in adjacent highway sectors (*SpeedLimitCalculator* and *RampSignalCalculator* tasks). The latter is expressed using the **`logical-hops`** construct relative to the **`HighwaySector`** attribute. Finally, data items describing the actions to perform are delivered to dedicated tasks instantiated on nodes equipped with the corresponding device, i.e., speed limit displays for the *SpeedLimitDisplayer*, and ramp signals for the *RampSignalDisplayer*.

The second application, depicted in Fig. 10, targets a *building environment management* system. Essentially, the processing is similar to the cluster-based data aggregation of Fig. 2, but now gathering data from two different types of sensors. The **`@TemperatureSensor`** and **`@HumiditySensor`** constructs are used to distinguish nodes with different types of sensing devices. Additionally, the cluster-head also outputs data items representing actions to perform on the environment. These items are input to an additional 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.

## *6.2 Evaluation of the Compiler*

**Code Functionality.** The logic for both applications was hand-coded to perform simulation studies on the underlying routing mechanisms [34]. The hand-written code also allowed to verify the functionality of the ATaG compiler, by comparing the automatically generated code with the one used in the aforementioned studies. Indeed, by comparing the simulation logs obtained using the SWANS/Jist [4] sim-

ulator, it was confirmed that the compiler-generated code is functionally equivalent to the hand-written version.

**Settings for Performance Studies.** Here we look at the *time* and *memory* taken to compile the above ATaG programs. Since the task firing model used 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, the following factors were seen to be 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.

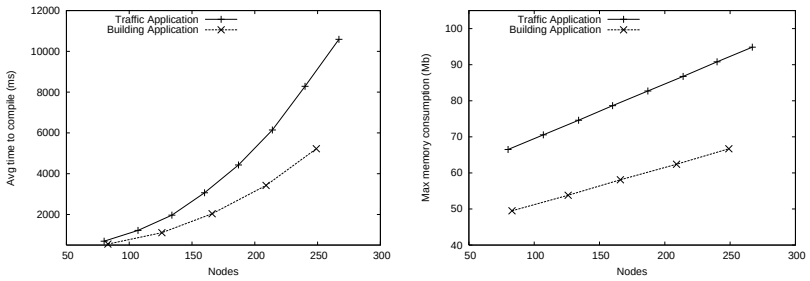
| | **Building** | **Traffic** |
|---|---|---|
| *Abstract Tasks* | 4 | 8 |
| nodes-per-instance:x@PLabel | 3 | 4 |
| partition-per-instance:x/PLabel | 1 | 4 |
| *Abstract Data Items* | 3 | 6 |
| *Abstract Channels* | 6 | 14 |
| local | 3 | 6 |
| domain | 3 | 4 |
| logical-hops:1(PLabel) | 0 | 4 |

**Fig. 11** Complexity of the task graphs of sample applications.

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 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. Fig. 11 reports the values of these factors seen in the sample applications.

In these tests, the compilation framework has been instantiated with the prototype implementations described in Section 4 for each module. In particular, the naive estimator and an *always-firing* task firing model were employed. For each test performed, the compilation process was repeated 500 times to account for fluctuations due to concurrent processes.

**Performance Results.** Fig. 12 illustrates the performance of the 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 the




(a) Time taken to compile.

(b) Maximum memory consumed during compilation.

**Fig. 12** Performance of the ATaG compiler w.r.t target network size.

naive estimator 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 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).

In exchange for the above costs in term of memory and time, the framework buys the developer *ease-of-use* in implementing the application using ATaG macro-programs, as discussed next.

## *6.3 Evaluation of the Toolkit*

To evaluate the performance of *Srijan*, both the applications discussed in Section 6.1 were developed using it. For each of the applications, the complete end-to-end development was performed – starting from specifying the ATaG task graph to deployment of code on the nodes – using *Srijan*. The developer used a Pentium-4 2.8 GHz laptop with 1GB of RAM running Windows XP for evaluation. The deployment was done onto the Sun SPOT [49] 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. The Java hProf profiler [23] was used for measuring execution time.

During the experiments, a variety of statistics was collected. 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, the experiment 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, the numbers are reported because **a)** these numbers are of the 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 the *Srijan* toolkit.

In addition to the above objective metrics, the experiment 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. Note that that these timings are variable from person to person, and more accurate statements can be made only after doing large user-studies.

| | **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 |

**Fig. 13** Costs involved in various stages of application development using *Srijan*

The experimental data is summarized in Table 13. 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, **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. Fig. 14 shows that the user-generated code is only around 2% of the total code. Even if the library code is neglected, *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

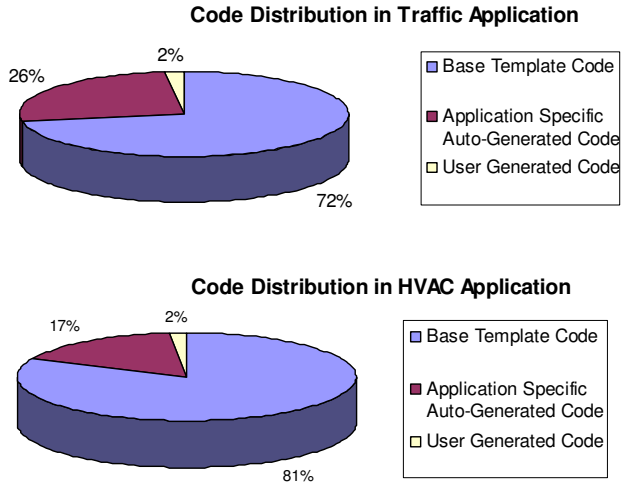



**Fig. 14** Distribution of code generation effort

it was not available as part of *Srijan*. These initial experiments demonstrate that the toolkit makes application development for WSNs more convenient for the domain expert.

## 7 Concluding Remarks

In spite of the developments in various areas of supporting applications on wireless sensor networks, the difficulty in application development still presents a hurdle to their wide acceptance. This chapter presented an overview of the various approaches available to WSN application developers for creating their applications, both at the node level and at the system-level. Focussing specifically on data-driven macroprogramming, we discussed a general compilation framework for a data-driven macroprogramming language for sensor networks. The framework was then shown to be used for developing a compiler that can convert macroprograms written in ATaG into a running sensor system, which was followed by a discussion of *Srijan* – a graphical toolkit for end-to-end development of WSN applications using the ATaG data-driven macroprogramming language. Through experiments, it was shown that the time taken to compile the macroprogram depends closely on the complexity of both the macroprogram and that of the target sensor system, and also that using *Srijan*, developers can quickly develop realistic WSN applications while writing a very small fraction of the actual application code.

Although data-driven macroprogramming aims to help make WSN application design easy, it is up to the designers of the compiler and runtime system to make up for the loss of performance (e.g., high energy costs, shorter lifetimes) that inevitably accompany a rise in the level of abstraction by incorporating optimizations in the

compilation process. Two such optimizations that can be looked at are placing tasks to reduce to communication costs and increase the system lifetime; and performing logical-expression sharing when combining the scopes in the channel-annotations of the task graph. We believe that future research in the domain will see experts in each stage of the compilation process developing better techniques and algorithms for their part.

**Acknowledgements** This work is partially supported by the National Science Foundation, USA, under grant number CCF-0430061 and CNS-0627028. The authors are grateful to Amol Bakshi, Luca Mottola, Gian-Pietro Picco, and Qunzhi Zhou for their assistance.

## References

1. Awan, A., Jagannathan, S., Grama, A.: Macroprogramming heterogeneous sensor networks using cosmos. In: EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 159–172. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1272996.1273014
2. Bakshi, A., Pathak, A., Prasanna, V.K.: System-level support for macroprogramming of networked sensing applications. In: Int. Conf. on Pervasive Systems and Computing (PSC) (2005)
3. Bakshi, A., Prasanna, V.K., Reich, J., Larner, D.: 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)
4. Barr, R., Haas, Z.J., van Renesse, R.: Jist: an efficient approach to simulation using virtual machines. Softw. Pract. Exper. **35**(6) (2005)
5. Bernat, G., Burns, A., Wellings, A.: Portable worst-case execution time analysis using java byte code. In: Proc. of the $12^{nd}$ Euromicro Conf. on Real-Time Systems (2000)
6. Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A., Riabov, A., Ye, F.: 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)
7. Boukerche, A.: Algorithms and Protocols for Wireless and Mobile Systems. Chapman & Hall/CRC 2005 (2005)
8. Boulis, A., Han, C.C., Srivastava, M.B.: Design and implementation of a framework for efficient and programmable sensor networks. In: MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services, pp. 187–200. ACM, New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/1066116.1066121
9. Buonadonna, P., Gay, D., Hellerstein, J., Hong, W., Madden, S.: TASK: sensor network in a box. In: Second European Workshop on Wireless Sensor Networks, EWSN 2005. (2005)
10. Chatzigiannakis, I., Mylonas, G., Nikoletseas, S.E.: jWebDust : A java-based generic application environment for wireless sensor networks. In: DCOSS, pp. 376–386 (2005)
11. Cheong, E., Lee, E.A., Zhao, Y.: Joint modeling and design of wireless networks and sensor node software. Tech. rep., Electrical Engineering and Computer Sciences University of California at Berkeley (2006)
12. Choi, W., Shah, P., Das, S.: 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)
13. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proc. of the $22^{nd}$ Int. Conf. on Software Engineering (ICSE) (2000)

14. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Tinylime: bridging mobile and sensor networks through middleware. In: Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on, pp. 61–72 (2005). URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1392743
15. Dermibas, M.: Wireless sensor networks for monitoring of large public buildings. Tech. rep., University at Buffalo (2005)
16. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proceedings of Programming Language Design and Implementation (PLDI) (2003)
17. The Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme
18. GRATIS: Graphical development environment for tinyos. http://www.isis.vanderbilt.edu/projects/nest/gratis/index.html
19. Habitat Monitoring on the Great Duck Island. www.greatduckisland.net
20. Greenstein, B., Kohler, E., Estrin, D.: A sensor network application construction kit (SNACK). In: 2nd ACM Conference on Embedded Networked Sensor Systems (2004)
21. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using Kairos. In: Proc. of the $1^{st}$ Int. Conf. on Distributed Computing in Sensor Systems (DCOSS) (2005)
22. Hnat, T.W., Sookoor, T.I., Hooimeijer, P., Weimer, W., Whitehouse, K.: Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In: SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems, pp. 225–238. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1460412.1460435
23. HPROF: A heap/cpu profiling tool in J2SE 5.0. http://java.sun.com/developer/technicalArticles/Programming/HPROF.html
24. Hsieh, T.T.: Using sensor networks for highway and traffic applications. IEEE Potentials **23**(2) (2004)
25. Jain, V., Biswas, R., Agrawal, D.P.: 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)
26. Karp, B., Kung, H.T.: GPSR: Greedy perimeter stateless routing for wireless networks. In: Proc. ACM/IEEE MobiCom (2000)
27. Kothari, N., Gummadi, R., Millstein, T., Govindan, R.: Reliable and efficient programming abstractions for wireless sensor networks. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, pp. 200–210. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1250734.1250757
28. Krishnamachari, B.: Networking Wireless Sensors. Cambridge University Press (2006)
29. Liu, J., Reich, J., Zhao, F.: State-centric programming for sensor-actuator network systems. IEEE Pervasive Computing (2003)
30. Luo, L., Abdelzaher, T.F., He, T., Stankovic, J.A.: Envirosuite: An environmentally immersive programming framework for sensor networks. Trans. on Embedded Computing Sys. **5**(3) (2006)
31. Luo, L., Cao, Q., Huang, C., Wang, L., Abdelzaher, T.F., Stankovic, J.A., Ward, M.: Design, implementation, and evaluation of enviromic: A storage-centric audio sensor network. ACM Trans. Sen. Netw. **5**(3), 1–35 (2009). DOI http://doi.acm.org/10.1145/1525856.1525860
32. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. **30**(1), 122–173 (2005). DOI http://doi.acm.org/10.1145/1061318.1061322
33. Mainland, G., Welsh, M., Morrisett, G.: Flask: A language for data-driven sensor network programs. In: Technical Report TR-13-06, Harvard University Technical Report (2006)
34. Mottola, L., Pathak, A., Bakshi, A., Prasanna, V.K., Picco, G.: Enabling Scoping in Sensor Network Macroprogramming. Technical report. Submitted for publication. Available at http://indus.usc.edu/atag (2006)
35. Mottola, L., Pathak, A., Bakshi, A., Prasanna, V.K., Picco, G.P.: Enabling Scoping in Sensor Network Macroprogramming. In: Fourth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS) (accepted) (2007)

36. Mottola, L., Picco, G.P.: 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)
37. Mottola, L., Picco, G.P.: 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)
38. Mottola, L., Picco, G.P.: Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Computing Surveys (accepted)
39. Newton, R., Arvind, Welsh, M.: 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)
40. Newton, R., Girod, L., Craig, M., Morrisett, G., Madden, S.: Design and evaluation of a compiler for embedded stream programs. In: Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008) (2008)
41. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Proc of the $1^{st}$ Int. Workshop on Data Management for Sensor Networks (DMSN) (2004)
42. Pathak, A., Mottola, L., Bakshi, A., Prasanna, V.K., Picco, G.P.: 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)
43. Pathak, A., Prasanna, V.K.: 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)
44. Pathak, A., Prasanna, V.K.: Energy-efficient task mapping for data-driven sensor network macroprogramming. In: International Conference on Distributed Computing in Sensor Systems (DCOSS) (2008)
45. Powell, O., Leone, P., Rolim, J.D.P.: Energy optimal data propagation in wireless sensor networks. J. Parallel Distrib. Comput. **67**(3), 302–317 (2007)
46. Rahimi, M., Hansen, M., Kaiser, W., Sukhatme, G., Estrin., D.: Adaptive sampling for environmental field estimation using robotic sensors. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2005)
47. Sookoor, T., Hnat, T., Hooimeijer, P., Weimer, W., Whitehouse, K.: Macrodebugging: global views of distributed program execution. In: SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, pp. 141–154. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1644038.1644053
48. *Srijan* - Graphical WSN application development toolkit. `https://gforge.inria.fr/projects/srijan/`
49. Sun$^{TM}$Small Programmable Object Technology (Sun SPOT), `www.sunspotworld.com`
50. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proc. of the $2^{nd}$ Int. Conf. on Mobile systems, applications, and services (MOBISYS) (2004)
51. Whitehouse, K., Zhao, F., , Liu, J.: Semantic streams: A framework for composable semantic interpretation of sensor data. In: European Workshop on Wireless Sensor Networks (EWSN) (2006)
52. Woo, A., Seth, S., Olson, T., Liu, J., Zhao, F.: 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). DOI http://doi.acm.org/10.1145/1127777.1127842
53. Wood, A.D., Selavo, L., Stankovic, J.A.: SenQ: An embedded query system for streaming data in heterogeneous interactive wireless sensor networks. In: S.E. Nikoletseas, B.S. Chlebus, D.B. Johnson, B. Krishnamachari (eds.) DCOSS, *Lecture Notes in Computer Science*, vol. 5067, pp. 531–543. Springer (2008)
54. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. **31**(3), 9–18 (2002). DOI http://doi.acm.org/10.1145/601858.601861