A Compilation Framework for Macroprogramming Networked Sensors*

Animesh Pathak[1], Luca Mottola[2], Amol Bakshi[1],
Viktor K. Prasanna[1] and Gian Pietro Picco[3]

[1] {animesh, amol, prasanna}@usc.edu,
Ming Hsieh Department of EE-Systems, University of Southern California, USA
[2] mottola@elet.polimi.it,
Dipartimento di Elettronica ed Informazione, Politecnico di Milano, Italy
[3] picco@dit.unitn.it,
Department of Information and Communication Technology, University of Trento, Italy

Abstract. Macroprogramming—the technique of specifying the behavior of the *system*, as opposed to the *constituent nodes*—provides application developers with high level abstractions that alleviate the programming burden in developing wireless sensor network (WSN) applications. However, as the semantic gap between macroprogramming abstractions and node-level code is considerably wider than in traditional programming, converting the high level specification to running code is a daunting process, and a major hurdle to the acceptance of macroprogramming.

In this paper, we propose a general compilation framework for a data-driven macroprogramming language that allows for plugging in different modules implementing various stages of compilation. We also demonstrate an actual instantiation of our framework by showing an end-to-end solution for compiling macroprograms. Our compiler provides the final code to be deployed on real nodes as well as an estimate of the costs the running system will incur, e.g., in terms of messages exchanged. We compared the auto-generated code against a hand-coded version for the same application behavior to verify the outcome of our compiler.

1 Introduction

Macroprogramming refers to a set of programming techniques whose objective is to increase application developers' productivity and allow non-expert programmers to write distributed, sense-and-respond applications easily. Abstractions are provided to specify the high-level collaborative behavior at the *system level*. Most of the low-level details concerning state maintenance or message passing are intentionally hidden 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 efforts [2, 11, 20] currently underway in this field.

* This work is partially supported by the European Union under the IST-004536 RUNES project and by the National Science Foundation, USA, under grant number CCF-0430061.

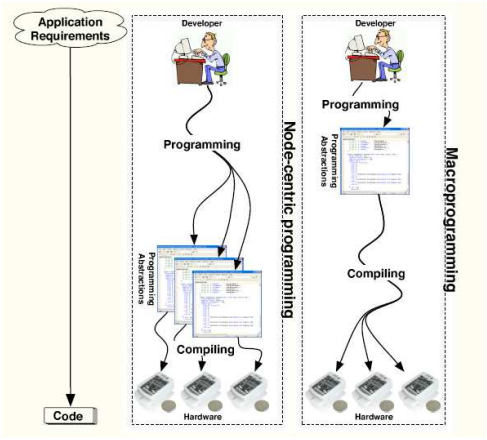


Fig. 1. Comparing node-centric and macroprogramming.

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 more difficult than in the node-level compilers currently seen in WSNs.

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 [22], we summarized the challenges faced by the designers of compilation frameworks for macroprogramming languages. As illustrated in Sect. 3, 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.

In this paper, we present the design, implementation and evaluation of a compilation framework to support macroprogramming. Specifically, we focus on a data-driven macroprogramming model called the *Abstract Task Graph* (ATaG) [2], whose salient features are described in Sect. 2. We make two *contributions* in this paper:

- We propose a general framework for compilation used for data-driven macroprogramming languages like ATaG. An overview of the compilation process is given in Sect. 3. 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. Our compilation framework is described in Sect. 4.
- 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. As described in Sect. 5, the

functionality of our compiler is assessed by inspecting and comparing the auto-generated code against a manually developed version of the same.

Compilation of macroprograms is still in its formative stages, and there is great variety in both the current work and future directions in the community. A discussion of related work is presented in Sect. 6. Section 7 concludes this paper.

2 ATaG: Abstract Task Graph

Macroprogramming of WSNs is an active area of research, with several programming paradigms currently being investigated [2, 11, 20]. In this work, we focus on ATaG (Abstract Task Graph) [2], a data-driven macroprogramming framework. ATaG includes an extensible, high-level *programming model* to specify the application behavior, and a corresponding node-level *run-time support*, called DART [1]. 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 specifi cation. 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.

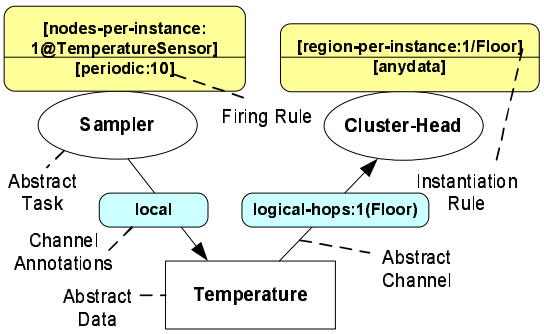


Fig. 2. ATaG program for data-gathering

The concept of *abstract data items* and *abstract tasks* are integral to specifying applications in ATaG. The former represents the information generated and communicated in the system, while the latter is a logical entity encapsulating the processing of one or more data items. The processing within a task is expressed using an imperative language. The flow of information between tasks is defi ned by *abstract channels*, which connect a task to a data item when the task *produces* that item, or vice versa when the task *consumes* it. Not that in an ATaG program, a data item can have more than one *consumers*, but only one *producer*.

Figure 2 illustrates an example ATaG program specifying a data gathering application [5] for building environment monitoring. Sensors within a cluster take periodic temperature readings, which are then collected by the corresponding cluster-head. The former aspect is encoded in the *Sampler* task, while the latter is represented by *Cluster-Head*. The *Temperature* data item is connected to both tasks using abstract channels.

Tasks are annotated with *firing* and *instantiation rules*. The former specify when the processing in a task must be triggered. In our example, the *Sampler* is triggered every 10 seconds according to the **`periodic`** rule. Differently, 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 instantiation rules govern the placement of tasks on real nodes, whose characteristics (e.g., sensing device attached) are encoded using node attributes. The **`nodes-per-instance:`**q**`@Device`** 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. Differently, the programmer requires a single *Cluster-Head* to be instantiated on every floor in the building. The **`partition-per-instance:1/Floor`** construct is used for this purpose. 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.

Abstract channels are annotated to express the *interest* of a task in a data item. In our example, 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

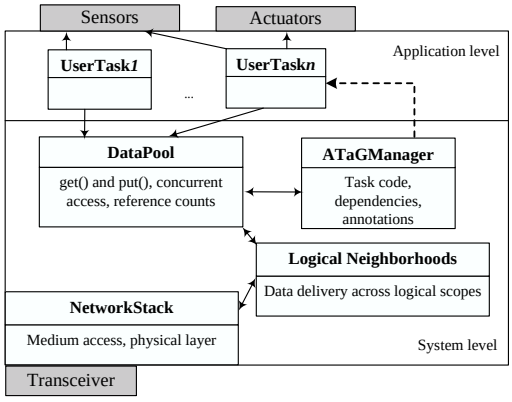


Fig. 3. DART: Data-driven ATaG run-time system.

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 depicts the architecture of our runtime system [1]. 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 *LogicalNeighborhoods* [17, 18] module handles data delivery 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 first *Floor* as intended recipients. Finally, the *NetworkStack* is in charge of communication with other nodes in the network, and manages 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 of Data-Driven Macroprograms: Overview

In this section, we provide an overview of the compilation process using the application given in Fig. 2 as example. 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 $\{AD_1 \rightarrow AT_2\}$. The compiler generates a set of node-level programs based on AT and the description N of the target system.

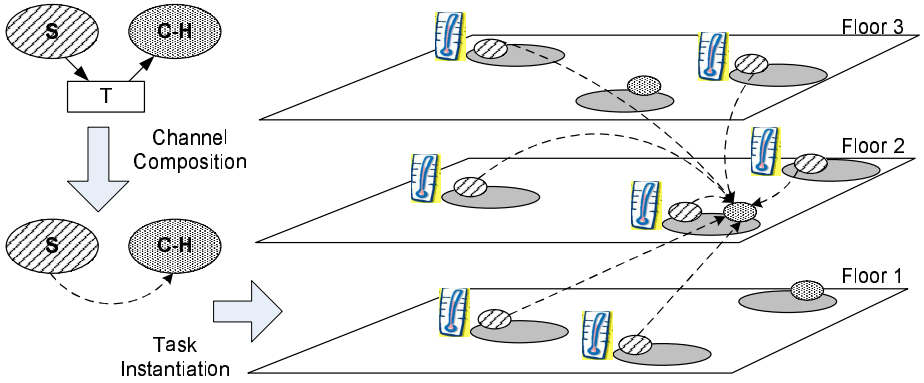


Fig. 4. An example illustrating the compilation process of our sample program.

Composition of 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 developer, traditional task allocation techniques work on task graphs with *direct depen-*

dency links between tasks. To address the problem of generating such task graphs, we convert each *path* $AT_i \rightarrow AD_k \rightarrow AT_j$ to an *edge* $AT_i \rightarrow AT_j$.

Since the channels in ATaG have logical scopes associated with them, composing two channels into one poses its own set of challenges. The basic 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* ± 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.

This task graph with composed channels is then instantiated on the given target network. Figure 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.

ITaG: Instantiated Task Graph. The intermediate representation used for applying task-allocation techniques is called the *instantiated task graph* (ITaG). It is a representation of the target system, with the tasks assigned to each node and communicating with each other. It consists of multiple copies of each abstract task specified in the ATaG program, each assigned to a particular node. The (directed) edges of the ITaG connect each task to the tasks that depend on it, i.e., the tasks that a) 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 $I(IT, IC)$ consists of a set IT of instantiated tasks and a set IC of instantiated channels. For each task AT_i in the ATaG 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 which node 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 analysing task graphs.

In our example, since there are seven nodes with attached temperature sensors, $f(AT_1, N) = 7$. 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.

In summary, the compiler is responsible for generating an efficient task placement, ensuring that the composed channels are consistent with the semantics specified by the application developer in the abstract channels, and configuring the runtime system modules. An added complexity in the compilation process is brought by the large space of

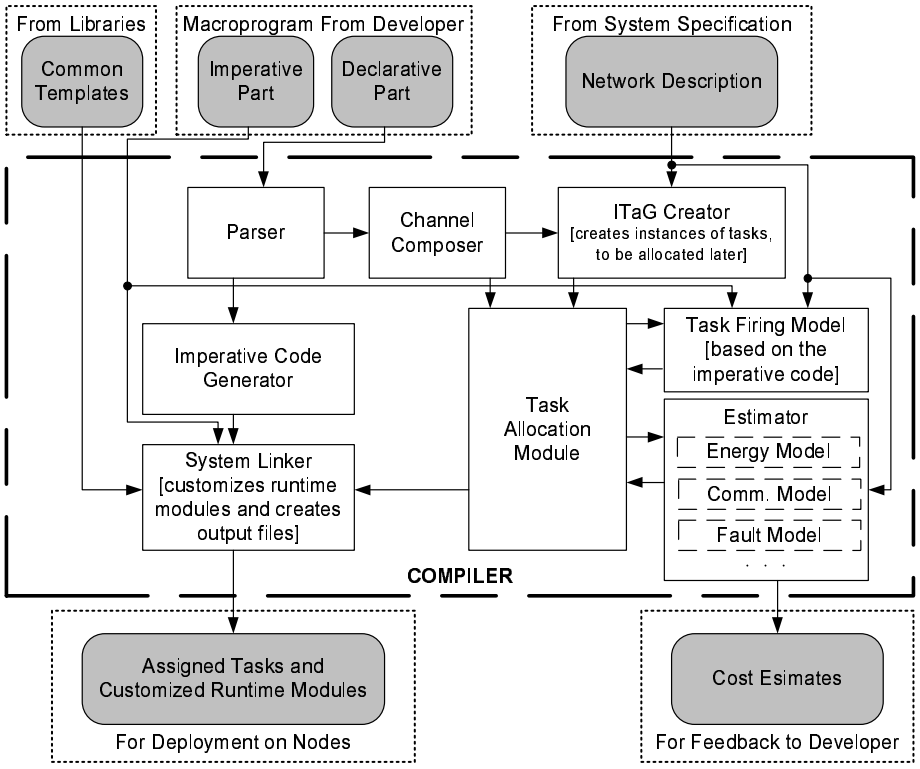


Fig. 5. The ATaG compilation framework.

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.

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, 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 first illustrate the input and output of our framework (illustrated in Fig. 5), and then proceed to the description of the different modules implementing the compilation itself.

4.1 Compilation Input and Output

The information provided at the beginning of a compilation effort are:

ATaG declarative specification: consisting of the abstract task graph itself, i.e., the set of abstract tasks and abstract data items, connected by abstract channels.

ATaG imperative code: namely, the description of the actions taken when each task is fired, expressed in an imperative language.

DART run-time templates: including both the node-level code later customized by the compiler, and generic supporting mechanisms, e.g., for routing messages.

Network description: containing information on the target deployment scenario, e.g., number, location and attributes of nodes. The attributes may contain information about the logical region the node is in (e.g., floor number), and the sensors or actuators attached to it. The need for a separate network description is dictated by ATaG's characteristic of being *deployment agnostic*. Since ATaG programs do not assume any specific target deployment, the program can be easily re-deployed if the target changes. Moreover, the network description does not necessarily include information on node connectivity. Depending on the constructs employed in the ATaG program, it may be sufficient to provide the list of target nodes along with the corresponding attributes exported.

The above input to the compilation framework is used to derive: (i) the **files to be deployed** on the real nodes, sorted according to the node identifier, and (ii) **cost estimates** to provide feedback to the application developer. Note that the actual nature of the cost estimates returned can vary depending on the developer 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.

4.2 Compilation Modules

We encapsulated the compilation stages we identified 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. Still referring to Fig. 5, we now describe these different modules, also pointing out the implementations we have realized so far.

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. Looking at 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 Sect. 3.

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. The ITaG creator then instantiates as many copies of the task as the product of the number of target regions and the number of instances per target region required in the ATaG program. 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 constitutes the core of the compilation process, since its job is to output 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 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, we currently employ a simple randomized 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. This is among our immediate research goals.

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 [3]: 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 [4, 6].

System Linker. At the end of the whole process, the linker module combines the information generated by the various paths of the compilation into the actual code to be deployed on the real nodes. More specifically, it configures the *ATaGManager* 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.

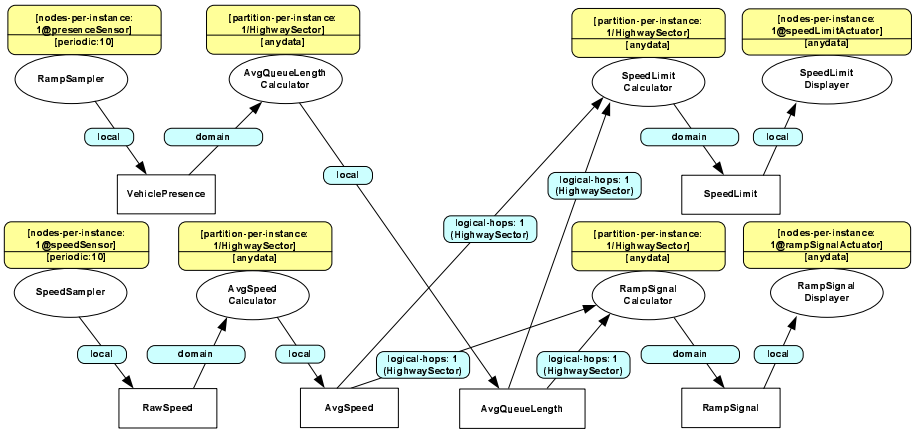


Fig. 6. An ATaG program for highway traffic management.

5 Demonstration

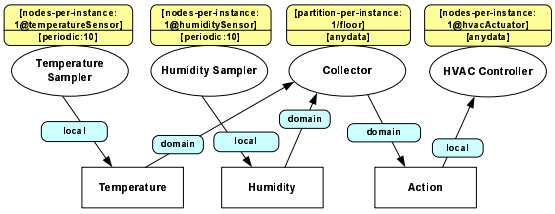


Fig. 7. An ATaG program for building environment management.

To demonstrate the effectiveness of our prototype 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. 6, 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 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 *AvgQueueLengthCalculator* tasks), and then delivered to tasks deciding the actions taken in adjacent highway sectors (*SpeedLimitCalculator* and *RampSignalCalculator* tasks). Note 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. 7, targets a *building environment management* system. Essentially, the processing is similar to the cluster-based data ag-

gregation 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.

Code Functionality. We hand-coded the logic for both applications to perform simulation studies on the underlying routing mechanisms [16]. 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 [3] simulator, we confirmed that the compiler-generated code is functionally equivalent to the hand-written version. The specific code samples can be found at [21].

	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

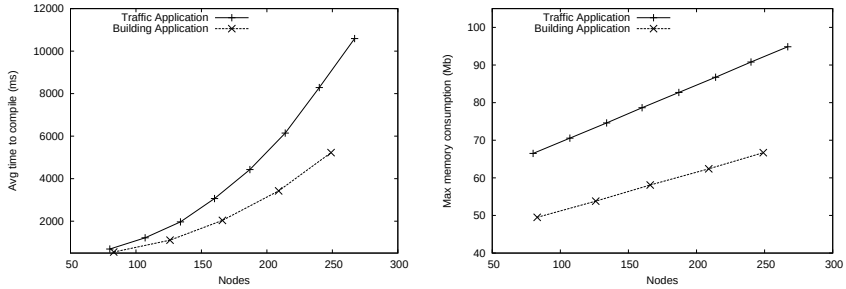
Fig. 8. Sample applications.

Settings for Performance Studies. We look at 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 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 8 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 for each module. In particular, we have chosen to employ the naive estimator and an *always-firing* task firing model. For each test we per-



(a) Time taken to compile.

(b) Maximum memory consumed during compilation.

Fig. 9. Compiler performance.

formed, 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 [7] profi ler.

Performance Results. Figure 9 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 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 traffi c 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 fi xed 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 macroprograms. To reassert this fact, we note that looking at the number of Java classes compiled to deploy our traffi c application on a single node, it turns out only 15 out of a total of 51 classes are the direct result of the developer's effort. The others are the implementation of the DART run-time system. Furthermore, considering the actual number of lines of non-commented code, only about 12% of the imperative code is hand-written by developers, whereas the rest is either part of the run-time support, or automatically generated.

6 Discussion

Initial programmming of WSNs was done by the nesC [8] language and the tinyOS operating system [12], and helped a wide research community build and test applications and system components for networked sensing [9, 13, 14, 23]. Over time, tools such as SNACK [10] were developed to support the programmers of such systems, and sensor nodes supporting more traditional programming languages such as Java have also emerged [24]. However, the compilers of all these languages are essentially node-level compilers, not very different from the common C compiler used on larger machines.

Various macroprogramming approaches have been proposed recently to alleviate the programming burden for WSN application developers [11, 20]. Since we are not aware of published work specifically detailing their compilation process, we compare our work with the issues we expect would be addressed by similar systems for these languages based on existing literature. *Kairos* [11] 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. 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.

Regiment [20] 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 [19], 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.

EnviroSuite [15] 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 essentially 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.

This paper does not claim to completely solve the problem of compilation of macroprograms for WSN applications. Our main focus is to present a clear set of subtasks involved in the process, and the interrelationships of the modules implementing them. We believe that this will contribute towards the achievement of two goals. By clearly identifying the modules, we can help researchers in the community attack the particular subtasks involved in compilation. Clearly, more efficient techniques are required to

provide the functionalities of the *Estimator*, *Task Firing Model*, and the *Task Allocation* module. Another issue that remains to be addressed is the possibility of *timing conflicts* among the tasks that are instantiated on a node, which is part of our future work. Further, by presenting a proof of concept implementation of the compiler, domain experts can begin to use the ATaG macroprogramming framework and provide us feedback on the language, the compiler as well as the runtime system. Although our current implementation runs on a simulator, the nature of the SWANS/Jist system is such that the same code can be run on actual nodes. We indend to present a demo of our approach on SunSPOT [24] nodes in the near future.

7 Concluding Remarks

In this paper, we presented a general compilation framework for a data-driven macroprogramming language for sensor networks. We demonstrated the feasibility of our approach by developing a compiler that can convert macroprograms written in ATaG into a running sensor system. Our experiments indicate that the time taken to compile the macroprogram depends closely on the complexity of both the macroprogram and that of the target sensor system.

Our compilation framework currently assumes a *static* network structure, which greatly limits the class of applications that we can address using this approach. Even in those applications, issues such as faults cannot be addressed by the current approach. Our immediate future work will involve exploring *on-line* task migration algorithms that can continually work for optimizing the task allocation, in addition to effi cient algorithms for ascertaining *good* initial task placements.

References

1. A. Bakshi, A. Pathak, and V. K. Prasanna. System-level support for macroprogramming of networked sensing applications. In *Int. Conf. on Pervasive Systems and Computing (PSC)*, 2005.
2. 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)*, June 2005.
3. R. Barr, Z. J. Haas, and R. van Renesse. Jist: an efficient approach to simulation using virtual machines. *Softw. Pract. Exper.*, 35(6), 2005.
4. 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.
5. W. Choi, P. Shah, and S. 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.
6. 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.
7. DJProf Java Profiler, `www.mcs.vuw.ac.nz/ djp/djprof/`.
8. 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.

9. Habitat Monitoring on the Great Duck Island. `www.greatisland.net`.
10. B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *2nd ACM Conference on Embedded Networked Sensor Systems*, 2004.
11. 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)*, June 2005.
12. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.
13. B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. ACM/IEEE MobiCom*, August 2000.
14. B. Krishnamachari. *Networking Wireless Sensors*. Cambridge University Press, 2006.
15. L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.
16. L. Mottola, A. Pathak, A. Bakshi, V. K. Prasanna, and G. Picco. Enabling Scoping in Sensor Network Macroprogramming. Technical report. Submitted for publication. Available at http://indus.usc.edu/atag, 2006.
17. 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.
18. 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.
19. 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.
20. 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.
21. A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco. Compiling macroprograms using the ATaG compilation framework, http://indus.usc.edu/atag. Technical report, University of Southern California, 2007.
22. A. Pathak and V. 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.
23. M. Rahimi, M. Hansen, W. Kaiser, G. 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.
24. Sun™ Small Programmable Object Technology (Sun SPOT), `www.sunspotworld.com`.