Enabling Scoping in Sensor Network Macroprogramming*

Luca Mottola[‡], Animesh Pathak[†], Amol Bakshi[†], Viktor K. Prasanna[†], and Gian Pietro Picco[#]
[‡]Politecnico di Milano, Italy, mottola@elet.polimi.it
[†]University of Southern California, USA, {animesh, amol, prasanna}@usc.edu
[#]University of Trento, Italy, picco@dit.unitn.it

Abstract

Wireless sensor networks are increasingly employed to develop sophisticated applications beyond simple data gathering. In these scenarios, *heterogeneous* nodes are deployed, and *multiple parallel activities* must be performed to achieve the application goals. Therefore, application developers require the ability to *partition the system* based on the node characteristics, and specify the interactions between different partitions to implement the processing germane to different activities.

Node-level programming abstractions for sensor networks have already tackled this problem by providing a notion of *scoping*. However, the level of abstraction achieved is still not suited to implementing non-trivial, large-scale applications. In this paper we demonstrate how the aforementioned issue can be addressed by enabling *scoping concepts* in *macroprogramming* for sensor networks. Using macroprogramming, developers reason at a higher level of abstraction, focusing on the behavior to be achieved by the system as a whole. By enabling scoping in macroprogramming, they can capture the essence of a significant class of distributed, embedded applications in a very concise manner. This extremely simplifies the development process, and increases the maintainability and re-usability of the resulting implementations.

1 Introduction

Initial deployments of wireless sensor networks focused on a single, system-wide goal, and featured fairly simple architectures and algorithms. Habitat monitoring [10], a widely cited example in this respect, can indeed be implemented using mostly *homogeneous* nodes, each running the *same* application code. In these scenarios, developers are required to describe fairly simple patterns of interactions, e.g., that of sensing and reporting a physical reading.

Recent technological advances and the consequent advent of more powerful sensor nodes [14] are, however, enabling the use of WSNs in increasingly sophisticated settings, from smart spaces [24] to monitoring and control in buildings [8]. These applications often involve *heterogeneous* nodes equipped with actuators to influence the environment, and their ultimate goal is usually obtained by composing different, *collaborating activities*. For instance, a road traffic management application [13] is usually designed to perform at least two different activities, e.g., controlling the speed of vehicles on the highway, and regulating access through the ramps leading to it. For this purpose, various types

*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.

of sensors are employed, and different devices are installed to influence the environment, e.g., speed limit displays and ramp signals. In this respect, a non-trivial example is further described in Section 2.

Developers of these applications face several common challenges, due to the presence of different types of nodes and several concurrent activities. In these scenarios, they need to specify different system partitions and express the interactions between them, so as to map the processing implementing the different activities to the "right" subset of nodes. To achieve this, different notions of *scoping* have been proposed in the sensor network literature, e.g., [19, 26, 27]. These generally refer to the ability of grouping, at a logical level, nodes satisfying specific application requirements. This approach actually addresses *some* of the needs arising in developing complex applications, by masking heterogeneity and providing support for expressing non-trivial communication patterns. However, the currently available abstractions supporting scoping are targeted to node-level programming frameworks. As a result, developers are still forced to handle low level aspects such as parsing received messages, as in [27].

Clearly, more powerful abstractions must be provided to let domain-experts without a strong computing background develop complex sensor network applications. An answer to this need can be found in the context of *macroprogramming* [4, 11, 21], where higher-level abstractions are provided focusing on the system as a whole, and many low-level details related to inter-node communication and coordination are hidden. However, most of the existing approaches in macroprogramming do not explicitly address heterogeneity, and do not allow the specification of different collaborating parallel activities. To address this issue, in this paper we present the following contributions:

- We propose in Section 4 a precise definition of *scoping* in sensor networks to provide a foundation for our work. Based on this, we take an existing macroprogramming model, whose salient features are described in Section 3, and introduce novel programming constructs to enable the definition of system partitions and express the interactions between them. Using this approach, we provide application developers with a *logical* layer on top of the underlying physical system, abstracting away the *physical* location of data and nodes, as illustrated in Figure 1. This represents an added value to the programming model that developers can easily take advantage of: the programming activity is naturally brought to a realm where the focus is on the *application goals and requirements*, rather than on the *system* where the final implementation runs. The specific constructs enabling this are illustrated in Section 5, along with examples of their use in a non-trivial application.

- We demonstrate the *feasibility* of our approach by developing an end-to-end programming framework in support of our programming model. This includes all the aspects of the implementation process, from the compilation of the macroprogram to an analysis of the system performance. The former problem is tackled in Section 6, where we illustrate how the programming constructs we are proposing can be compiled down to node-level code that uses a dedicated run-time layer. The latter is discussed in Section 7, by showing simulation results obtained by running the actual code resulting from the aforementioned compilation process.

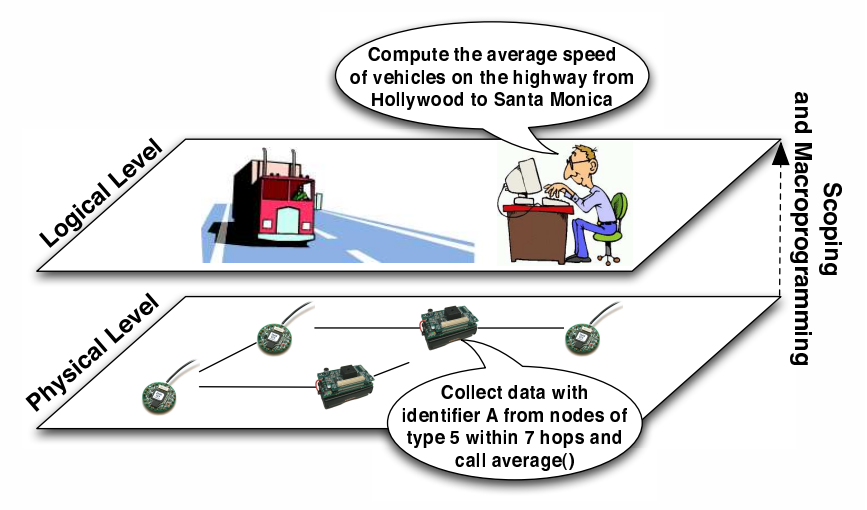


Figure 1: Raising the level of abstraction from the physical level to a logical level where only application data is exposed.

To discuss the advantages brought by scoping to macroprogramming, Section 8 reports on code metrics we gathered on the actual implementation of our reference application, and compares our programming model against existing proposals to highlight the scenarios where a specific model gives the greatest advantages. Section 9 provides brief concluding remarks and directions for future work.

The following section describes the reference application we have chosen for this study. While constituting a meaningful example on its own, it is also representative of a much larger class of applications, as it embodies many typical interaction patterns of non-trivial scenarios.

2 Reference Application

To showcase the complexity of the scenarios we target, here we consider a *road traffic monitoring and control* application, a field where WSNs have gained increasing attention from the research community [13]. Indeed, various techniques exist to influence the vehicle movement and improve traffic efficiency. These can be applied in various settings, ranging from metropolitan areas to highways. In the latter case, two of the most commonly used solutions are speed signaling [2] and ramp metering [16]. The former aims to control the behavior of traffic by suggesting appropriate speeds, while the latter influences traffic by controlling access to the highway. In these fields, different proposals exist to optimize goals such as pollution and fuel consumption [17].

Our reference scenario is depicted in Figure 2. Usually, this kind of system is divided into disjoint *sectors* [17], with each sector usually being controlled depending on the current status of the *same* and *neighboring* sectors. In the highway scenario we consider, a sector is identified by a single ramp leading to the highway, i.e., it spans the portion

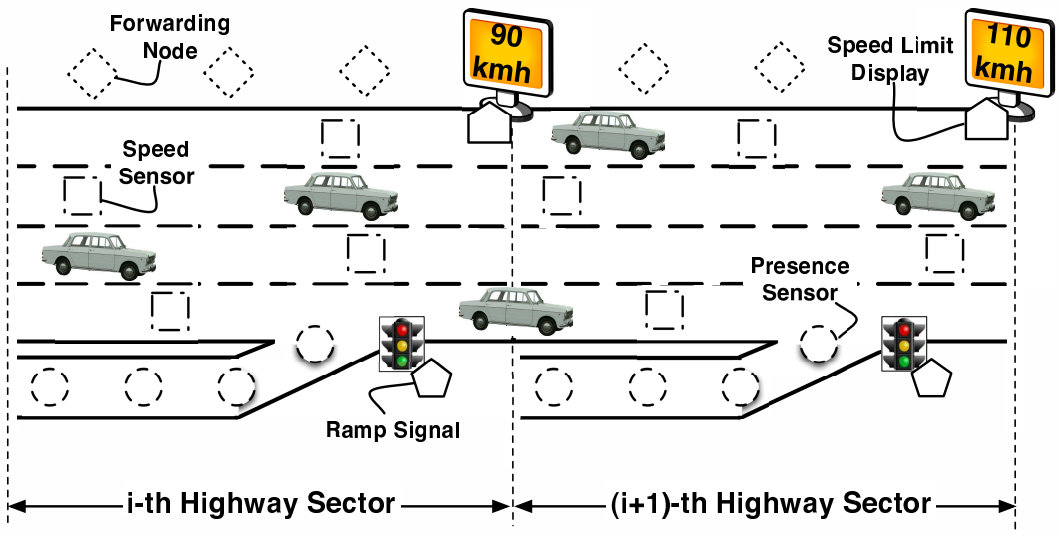


Figure 2: Scenario for the traffic management application.

of highway from a ramp to the following. The system has five main components:

- **Speed sensors** to measure and report the speeds of vehicles. They are installed on the lanes of the highway.
- **Presence sensors** to measure and report the presence of vehicles. They are installed on the ramps leading to the highway.
- **Speed limit displays** to inform the drivers of the recommended speed limit. They are installed on the road side, one per highway sector.
- **Ramp signals** designed to allow or disallow cars onto the highway. They are installed one per highway ramp.
- **Forwarding nodes** to enable wireless communication between the various nodes. They are installed on the road side at regular intervals.

Figure 3 illustrates, from a high-level perspective, the various stages of data processing in the application. Data is first collected from the sensing devices, and a first processing is performed to derive an aggregate measure such as the average speed of vehicles in a highway sector or the average queue length on a ramp. This information is fed as input to an algorithm determining the best actions to achieve the system objectives, e.g., to maximize the flow of vehicles on the highway. These actions are then communicated to the ramp signals and to the speed limit displays. The specific algorithms employed depend on the goals and metrics of interests. Therefore, a modular approach to the development of this class of applications may be advisable.

The application described above encapsulates behaviors and interactions seen in a large class of networked embedded applications [22]. These common characteristics are grounded in the use of *heterogeneous* nodes, and in the

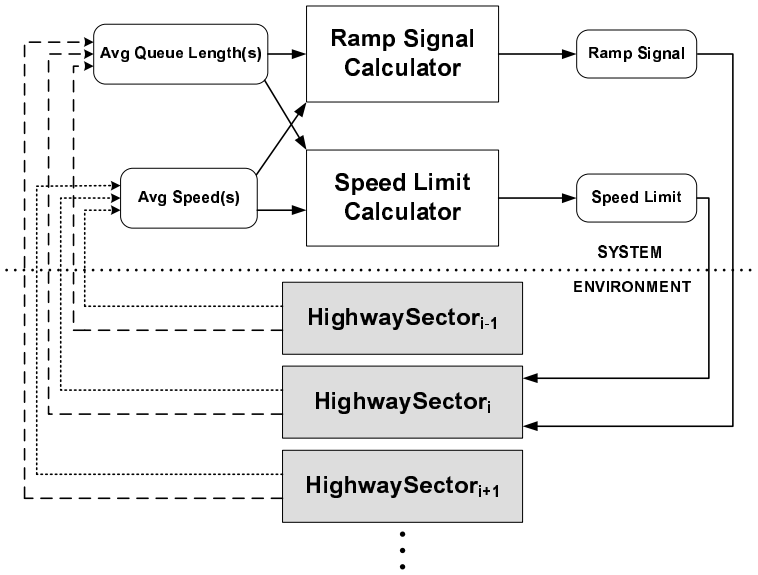


Figure 3: Data processing in traffic management.

presence of *multiple, concurrent activities* collaborating to achieve the application goals. Developers of these applications must therefore address requirements such as:

- **Multi-stage data processing:** as the raw sensor data is not useful by itself, the system needs to compute the average speed and queue length used to compute the ramp signals and speed limits. This represents a common need in sensor networks when actuation is involved [1].

- **Multiple sub-goals:** to achieve the high-level application objective, e.g., maximize the vehicle throughput on the highway, the system is required to run multiple parallel activities. In our case, regulating the speed of vehicles on the highway and controlling the access to it. This is often required when the system is designed to react to sensed data. For instance, in a different scenario like building monitoring and control, the system is normally required to perform at least three activities [7]: i) indoor environmental monitoring, ii) structural monitoring, and iii) response to extreme events such as fire.

- **Localized interactions:** each of the aforementioned sub-goals usually involves only a specific part of the system. For instance, controlling the speed in a specific highway sector relates to the sensors deployed on the lanes of three neighboring sectors only. Keeping the processing close to where data is sensed has been long recognized as an effective approach to save energy and achieve more efficient implementations [1, 9].

- **Heterogeneity handling:** various types of nodes are to be employed, with different characteristics and various devices attached. In our scenario, presence and speed sensors are employed along with nodes controlling the speed limit displays and ramp signals. Similarly, in building control and monitoring different kind of sensors

are used as well, e.g., temperature, humidity, and smoke sensors [8],

Most of the existing WSN programming frameworks cannot meet the above requirements easily. In first place, they do not provide programming constructs to enable a clear modularization of different activities or consecutive stages of processing. As a result, breaking the high-level application goal into smaller collaborating activities becomes hard to achieve. More importantly, they do not provide support for *heterogeneity*. It is therefore difficult to identify the portion of the system concerned with a specific activity. For instance, developers cannot map a specific processing to the nodes equipped with a given sensing device. These aspects are better discussed in Section 8, where we compare our work with existing approaches.

In this work we rely on *scoping* to give application developers a tool to address the aforementioned issues. This notion adequately provides the ability to partition the system depending on the application needs. Unlike existing work enabling some notion of scoping at the node-level, in this work we make scoping available to the application programmers at a high-level of abstraction, by enabling this concept in an existing macroprogramming model. This is illustrated next.

3 ATaG: a Macroprogramming Framework for Sensor Networks

Several efforts are currently underway in macroprogramming for sensor networks, e.g., [11, 21]. As a concrete illustration of our ideas, we enable scoping in the Abstract Task Graph [4] (ATaG), a macroprogramming framework providing a mixed *declarative-imperative* approach to the development of sensor network applications. It is characterized by two features: the first is *data driven* computing, which provides a natural model for specifying reactive behaviors, and the second is the use of *declarative specifications* to express the placement of processing locations and the patterns of interactions. An ATaG program, composed of its imperative and declarative parts, is given as input to a compiler, along with the list of physical nodes employed. This translates the high-level, abstract specifications in terms of the API provided by an underlying, node-level supporting run-time.

Programming Model. The notions of *abstract task* and *abstract data item* are at the core of ATaG's programming model. The former is a logical entity encapsulating the processing of one or more data items, representing the information itself. The flow of information between tasks is defined in terms of their input/output relations. To achieve this, *abstract channels* are used to connect a task to a data item when the task *produces* that item, or vice versa when the task *consumes* it.

Figure 4 illustrates an example ATaG program, specifying a simplified cluster-based, data gathering application [6, 12]. 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

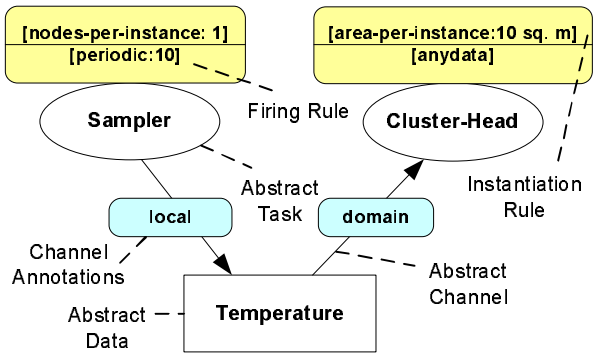


Figure 4: A sample ATaG program.

Temperature data item is connected to both tasks using a channel originating from the *Sampler* task, and a channel directed to *Cluster-Head*.

Tasks are annotated with *firing* and *instantiation rules*. The former specify when the processing in a task must be triggered. In the our example, the *Sampler* task is triggered every 10 seconds according to the **`periodic`** rule. Differently, the **`any-data`** firing rule requires *Cluster-Head* to run when at least one data item is ready to be consumed on *any* of its incoming channels. The instantiation rules govern the placement of tasks on real nodes. The **`nodes-per-instance:`**q construct requires the task to be instantiated once every q nodes. As $q = 1$ in the example, the *Sampler* task is instantiated on every node. The **`area-per-instance`** construct used for *Cluster-Head* implies partitioning the geographical space and deploying *one* instance of the task per 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* uses the **`domain`** annotation to gather data from the temperature sensors in its cluster. This binds to some system partitioning, e.g., that obtained by **`area-per-instance`**, and connects the tasks running in the same partition.

The code within a task is the only imperative part in an ATaG program. To express the flow of information between tasks in the imperative code, programmers are provided with the abstraction of a *shared data pool*, where each task can *output* data, or be *notified* when some data of interest is available. To support the former aspect, a single `putData(DataItem)` operation is made available. The second aspect is handled by providing the programmer with an automatically generated template for each task, that lists an empty `handleDataItem()` function for each incoming channel. The programmer fills these functions implementing the processing associated to each input data item.

The notion of *task* in ATaG provides a powerful concept to modularize different stages of processing. In addition, the separation of declarative and imperative parts in an ATaG program, and a modular run-time system and compilation framework make the model easily extensible. Leveraging off these features, we devised novel annotations and

constructs to enable scoping in ATaG. This notion is defined next.

4 Scoping

In this section, we describe a precise definition of scoping in sensor networks, aiming to provide a solid basis for understanding. The actual constructs enabling the definition of scopes in the ATaG programming language will be discussed next.

A *scope* in WSNs can be informally defined as a *subset of nodes sharing similar characteristics or goals*[1]. In this work, we specify this notion using a *membership function* f, whose goal is that of determining the subset of nodes included in a scope. Specifically, we define the membership function as $f_{s,i}(j)$, where i is the node where f is evaluated, and j is the node whose membership in scope s must be determined. The boolean output of the function returns whether j is part of scope s for node i or not. The actual definition of $f_{s,i}$ is obtained as the composition of atomic *boolean predicates* on the nodes characteristics (called *node attributes* hereafter). For instance, a node attribute may describe the sensing devices attached to a node, and a predicate on that attribute may check whether a particular sensor is among them.

Two orthogonal dimensions combine to form a scope definition. We say a boolean predicate $p(\cdot)$ is *symmetric* when it does not depend on i, i.e., it is not a function of the node where the scope is evaluated. For instance, the predicate $hasSpeedSensor(j)$, returning whether j is equipped with a speed sensor, is a symmetric one. Therefore, a scope defined as $f_{s,i}(j) ::= hasSpeedSensor(j)$ will determine the same subset of nodes regardless of the particular node i where f is evaluated. Conversely, a predicate is said to be *asymmetric* when it does depend on i, as in $isSameSector(i, j)$. Thus, in our scenario a scope $f_{s,i}(j) ::= isSameSector(i, j)$ will return a different subset of nodes depending on the sector where i is installed.

In the general case, the membership function defining a scope is likely to be a combination of a symmetric part with an asymmetric one, as illustrated in Figure 5. For instance, in our reference application a node in each sector might define a scope to identify the nodes sensing the speed of vehicles in that sector, and gather data from them to evaluate the average measure. These nodes are those i) equipped with a speed sensors, and ii) installed in the same sector as the node requiring their readings. For this purpose, we must use the combination of a symmetric predicate —used to express "the nodes having a speed sensor"— with an asymmetric one —to describe "installed in the same sector".

[1] Hereafter, we will term *node* the hardware hosting CPU and main memory, whereas we will generally indicate as *device* the sensors or actuators attached to a node.

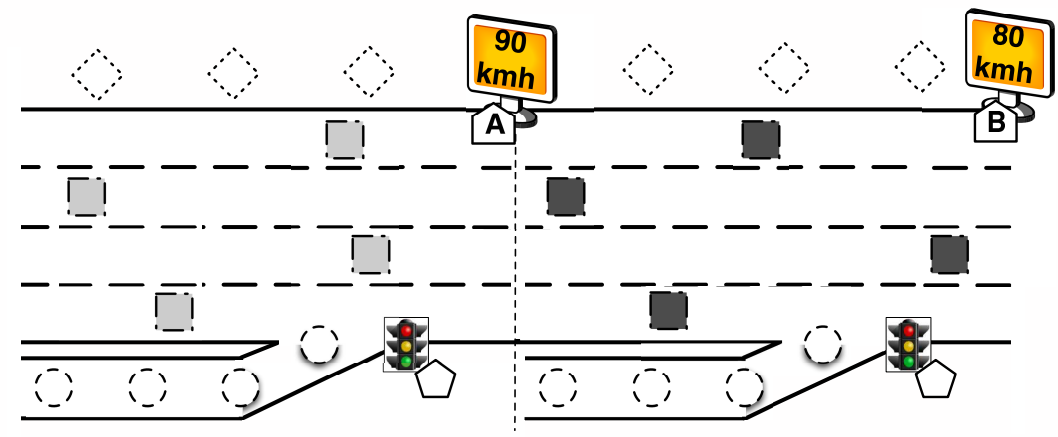


Figure 5: Scopes as the combination of symmetric and asymmetric predicates. Squared nodes (regardless of their coloring) are those satisfying a *symmetric* predicate to check whether they are equipped with a speed sensor, i.e., the set $\{j \in \mathcal{N} | hasSpeedSensor(j)\}$ ($\mathcal{N}$ being the set of nodes in the system). Light grey nodes are those included in scope $s1$, defined as $f_{s1,A} ::= isSameSector(A, j) \wedge hasSpeedSensor(j)$ and evaluated on node A. Symmetrically, dark grey nodes are those in scope $s2$, defined as $f_{s1,B} ::= isSameSector(B, j) \wedge hasSpeedSensor(j)$.

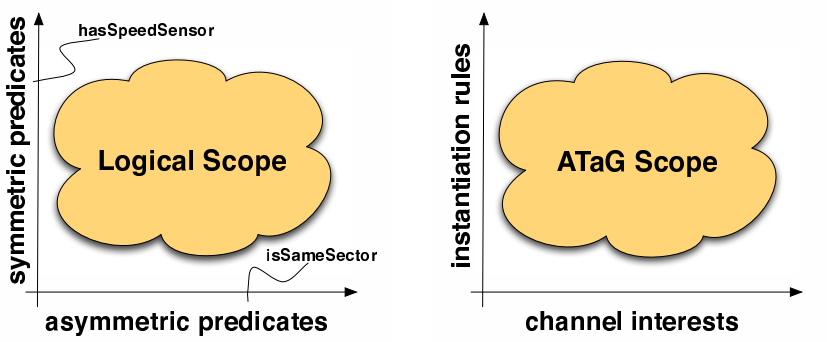
5 Scoping in a Macroprogramming Language

In this section, we provide an overview of how the aforementioned notion of scoping enhances the ATaG programming model, and then discuss the details of the specific programming constructs using an ATaG-based implementation of our reference application as example.

5.1 Overview

Augmenting the ATaG programming model with scoping affect primarily two aspects: *task placement* and *data exchange* between tasks. In the former case, scoping addresses *heterogeneity* among the nodes and the need for dividing the system into *logical regions*. Notably, an ATaG-based implementation of our reference scenario requires tasks to be instantiated on nodes equipped with the needed sensing/acting devices, or in specific regions only. For instance, a task designed to operate the ramp signal must be instantiated on a node having that particular device attached. Further, we need only one task to compute the average speed for each highway sector, so we need to identify the different sectors uniquely. This has been achieved with novel *instantiation rules*, that give application programmers the ability to define subsets of nodes satisfying specific constraints, e.g., that of being installed in the same highway sector. The ATaG compiler is instructed to instantiate a task on one or more nodes in a specific subset only.

As for data exchange between tasks, scoping is used by the developer to express the interactions among subset of nodes. In this sense, it enables the specification of *localized* interactions, as well as relations between the *multiple stages* of a given processing or multiple collaborating *sub-goals*. For instance, in our scenario the speed limit is decided



(a) A logical scope as the combination of symmetric and asymmetric predicates.

(b) A scope in ATaG as the combination of instantiation rules and channel annotations.

Figure 6: Embedding scoping in ATaG.

based on the information gathered from three neighboring highway sectors. To express this, we define new *channel interests* in ATaG, so that application programmers can specify the task interests by referring to the logical properties of data, regardless of their physical location. This specification is passed to the run-time support, that retrieves the data accordingly.

The combination of the novel instantiation rules and channel interests can be mapped to the two orthogonal dimensions we relied on to define scopes, as illustrated in Figure 6. Instantiation rules define subsets of nodes with common characteristics, e.g., having a particular actuator attached. As such, the subset they define is the same regardless of the node where it is evaluated, and can therefore be described with one or more *symmetric* predicates. Conversely, channel interests are typically described in terms of *asymmetric* predicates. They strictly depend on the associated task, therefore, the subset of nodes the application is interested in is a function of the node where the task is running. For example, in our scenario, the three neighboring sectors are relative to the particular sector where the node requesting the data for processing is located. This mapping is at the core of the translation process that generates the actual scope definitions from the ATaG constructs, described in Section 6.

5.2 ATaG Constructs for Scoping

The syntax and use of our scoping constructs are shown in Figure 7, where we illustrate the ATaG specification of the application described in Section 2. All the application information is represented as ATaG data items. The actual algorithm determining the actuation part is encapsulated in two tasks: *SpeedLimitCalculator* and *RampSignalCalculator*, whose inputs are the data produced by tasks deriving the average measures. Once the actuation is determined, this is given as input to the tasks operating the displays and ramp signals.

Task Placement. The *SpeedSampler* task is in charge of gathering the raw data from a speed sensor on a ramp leading to the highway. Therefore, it must run on a node equipped with the corresponding sensing device. To express

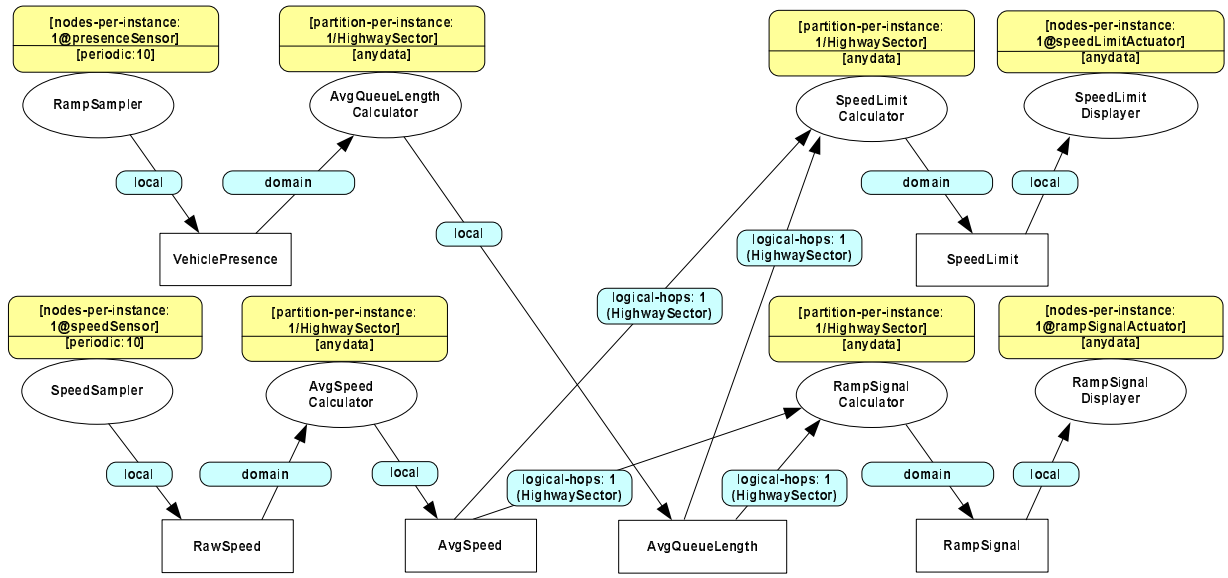


Figure 7: The ATaG program for the traffic management application.

```
<task name="SpeedSampler">
  <instantiationrule>
    <nodes-per-instance
           number="1"
           requiredAttributeType="AttachedSensors"/>
    <attribute type="AttachedSensors">
      <value="speedSensor">
    </attribute>
  </instantiationrule>
</task>
```

Figure 8: XML declaration for **`@speedSensor`** in Figure 7. (`AttachedSensors` is defined in a separate XML file listing the relevant attributes for each node).

this requirement, the **`nodes-per-instance:1@speedSensor`** construct is used, where **`@speedSensor`** is a placeholder for a boolean predicate determining the set of nodes equipped with a specific sensing devices. In our current prototype, the actual predicate is specified using a simple XML file, shown in Figure 8[2]. Similar constructs are used for *RampSampler*, *SpeedLimitDisplayer*, and *RampSignalDisplayer*.

The *AvgSpeedCalculator* task takes as input the raw data coming from the speed sensors in a sector, and derives the average speed of vehicles in the same sector. Therefore, we need such a task to be instantiated once per sector. To express this, the **`partition-per-instance:1/HighwaySector`** construct is used. This is based on the enumeration of possible values of the node attribute **`HighwaySector`** —that describes where a node is placed in the highway— and requires the task to be instantiated on one node in each sector only.

Data Exchange. To bind tasks running in the same **`HighwaySector`**, the **`domain`** annotation on a channel can be used. However, this time it is based on the system partitioning obtained through the **`partition-per-instance`**

[2]It is not our intention to force the programmer to write XML directly, we instead envision these specification to be auto-generated by an integrated development environment.

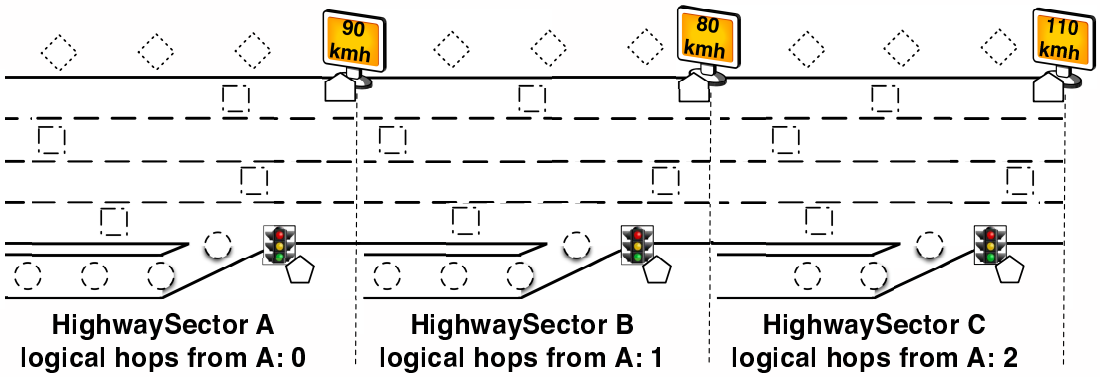


Figure 9: Logical hops over the **`HighwaySector`** attribute. The picture shows the number of logical hops to be crossed for a node in highway sector **`A`** to reach any node in a different highway sector.

instantiation rule. Differently from **`area-per-instance`**, this rule determines the different partitions at a logical level, by considering the node attributes instead of the geographical position.

In addition, the construct **`logical-hops:1(HighwaySector)`** connecting, e.g., the *AvgSpeedCalculator* to both the *SpeedLimitCalculator* and the *RampSignalCalculator* is used to collect a data item from different highway sectors. It represents a number of hops counted not on the physical network links, but in terms of how many system partitions (derived from the attribute given in parenthesis) can be crossed. Figure 9 illustrates the concept graphically. Given the partitioning induced by the **`HighwaySector`** attribute, requiring one logical hop on that attribute means, for a *SpeedLimitCalculator*, to collect a data item from the same, immediately preceding and following highway sectors. Notice how the semantics of specifying a number of zero hops is not to cross any partition, i.e., to collect from the same partition where the data item originated. In this sense, the **`domain`** construct actually constitutes a particular case of the more general **`logical-hops`** construct.

6 Compiler and Run-time Support

To enable scoping in a macroprogramming language, one needs to implement compiler support for the scoping constructs, and have an underlying, node-level run-time capable of providing data delivery to/from nodes in a given scope. It is indeed unreasonable to ask the compiler to generate the node-level code up to the network layers [20, 23].

In the prototype system we developed, we use the Java2ME [15] language and APIs to describe the imperative part of an ATaG program, targeting the upcoming SunSpot sensor platform [25]. As for supporting node-level run-time, we developed a Java version of Logical Neighborhoods [18, 19], a middleware-level programming abstraction providing a notion of scoping in WSNs. With Logical Neighborhoods, the physical neighborhood of a node is replaced by a logical notion of proximity determined by applicative information. A (logical) neighborhood is specified in a neighborhood *template*, that encodes a boolean predicate acting as a selection predicate over the set of possible nodes. For instance,

	ATaG	Logical Neighborhoods
Symmetric Predicates	Instantiation Rules	Neighborhood Template
Asymmetric Predicates	Channel Interests	Neighborhood Instantiation

Figure 10: Mapping scoping in ATaG to Logical Neighborhoods.

the $hasSpeedSensor$ predicate described in Section 4 can be inserted in a neighborhood template. The template is then *instantiated* on a specific node, by specifying where the encoded predicate must be evaluated w.r.t. the instantiating node. This is used to limit the span of the logical neighborhood, e.g., by specifying a maximum number of logical hops away from the instantiating node. Originally, the Logical Neighborhood run-time did not provide a way of specifying logical hops. However, adding this feature did not require any major modification in the processing required to determine the neighborhood members. In this case, the node attribute over which the logical hops should be counted must also be provided.

The neighborhood definition is fed to Logical Neighborhoods in the form of a suitable data structure. To interact with the nodes in a (logical) neighborhood, the programmer is provided with a simple message-passing API, used to *broadcast* (in a logical sense) a message to all nodes member of a neighborhood. An efficient routing scheme is provided in support of this API, illustrated in [19].

Given the logical neighborhood API, it is straightforward for the ATaG compiler to map scopes in ATaG to logical neighborhoods, as Figure 10 illustrates. Instantiation rules can indeed be considered as selection predicates over the set of nodes in the system, and are directly translated as neighborhood templates. Instead, the channel annotations actually constrain the span of the scope associated to a given channel. As such, they depend on the node where the task is running, and can therefore be translated as neighborhood instantiations on the same node.

For instance, the node where *AvgSpeedCalculator* is running gathers data output by *SpeedSampler* tasks in its same domain (highway sector). These are instantiated on nodes equipped with the corresponding sensor. Therefore, the compiler determines the nodes j from which *AvgSpeedCalculator* should gather data as those satisfying:

$$f_{s1,i}(j) \quad ::= \quad isSameSector(i,j) \wedge \\ hasSpeedSensor(j) \tag{1}$$

where i is the node where *AvgSpeedCalculator* is running. The latter conjunct is derived from the instantiation rule specified for the producer task (**`@speedSensor`**), and can therefore be specified as part of a neighborhood template. The former conjunct is instead derived from the channel interest (**`domain`**) and can be expressed at the time of instantiating the neighborhood on the node where *AvgSpeedCalculator* is running. Similarly, consider the tasks producing the data triggering the execution of *SpeedLimitCalculator*. In this case, the producer task can either be *AverageQueueLengthCalculator* or *AvgSpeedCalculator*, and can either be located on a node in the same sector, or in adjacent ones.

Therefore, the set of nodes j included in the scope is the one satisfying:

$$\begin{aligned} f_{s2,i}(j) \quad ::= \quad & isWithinNSectors(i, j, 1) \wedge \\ & (isRunningAvgSpeedCalculator(j) \vee \\ & isRunningAvgQueueLengthCalculator(j)) \end{aligned} \tag{2}$$

where *SpeedLimitCalculator* is running at node i. The first conjunct is again derived from the channel interest, (**`logical-hops:1(HighwaySector)`**), and is therefore specified as part of the neighborhood instantiation on node i, whereas the second determines the node included in scope $s2$ based on the task it is running, and is hence specified in a neighborhood template. The latter conjunct is needed since the instantiation rule used in this case does not uniquely specify the node where the producer task is running.

The compiler takes as input the list of nodes with their attributes, and the ATaG program. The compilation is carried out in a four step process:

1. The compiler allocates tasks to nodes by looking at the instantiation rules specified in the ATaG program, and matching them against the node attributes. Consistency checks are performed to ensure all the requirements on task placement can be satisfied.

2. Once the tasks are placed, the compiler identifies a set of *data paths* between nodes running tasks connected by some input/output relation. For each such path, the compiler combines the instantiation rules of the connected tasks with the channel annotations, and derives an abstract scope specification.

3. Given the abstract scope specifications, these are translated into neighborhood templates and neighborhoods instantiations, and given as input to the run-time support layer of each target node.

4. Additionally, the ATaG compiler configures other helper components in the run-time support layer [3].

When the instantiation rule does not uniquely specify the node where to instantiate a task, as in the case of **`partition-per-instance`**, the compiler currently places the task so that it is colocated with the producer task of at least one data item it consumes.

7 System Evaluation

To assess the feasibility of our approach, it is necessary to look at the performance of the running system. To that end, we run our traffic management application in a simulated scenario, and gathered performance metrics to characterize the system behavior. To do so, we used the SWANS/Jist simulator [5], as it is able to run unmodified Java code on top

Parameter Name	Value
Propagation Model	Two-ray Ground
Radio Model	Additive Noise
MAC Layer	CSMA
Transmission Rate	250 Kbps
Communication Range	40 meters
Message Size	47 bytes
Simulation Time	2000 secs
Number of Repetitions	30

Figure 11: Simulation parameters.

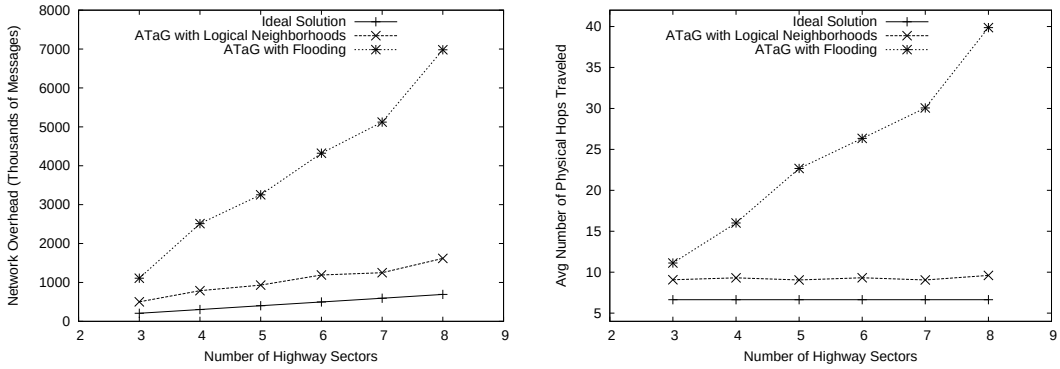
of a simulated network. This way, we measured the performance of the same code that can be deployed on the real nodes.

The relevant simulation parameters are summarized in Figure 11. As for network topology, we simulated the scenario represented in Figure 2 with a highway sector being 20 meters wide and 200 meters in length. We placed the forwarding nodes 25 meters apart, and randomly distributed the speed sensors on the four lanes so that each of them is range of at least another speed sensor or a forwarding node. Similarly, the presence sensors have been randomly distributed on the ramp so that each of them is in range of at least one speed sensor or another presence sensor. The node controlling the ramp signal and the speed limit display are placed at the border between different sectors, on the opposite sides of the road. Overall, 18 nodes are deployed in each highway sector. Also note the message rate is implicitly determined by the application itself, in particular by the firing rules for tasks. For instance, a node running an instance of *RampSampler* will generate one message every 10 seconds, as the corresponding firing rule is **`periodic:`**10. The *AvgQueueLengthCalculator* fires for any data item received, and correspondingly outputs a new data item. Therefore, if four *RampSampler* tasks are in its **`domain`**, the node running the *AvgQueueLengthCalculator* will generate a message every 2.5 seconds, on the average.

The various simulation runs differ in the initial random seed, in the location of nodes, and in the placement of tasks not tied to the node capabilities when more than a choice is available. As performance metrics, we consider the following:

- the number of *missing actuations* on the environment, resulting from *one or more message losses* on the path from the nodes running the sensing tasks to the nodes running the actuation tasks,
- the *network overhead*, represented as the overall number of messages sent at the physical layer,
- the average *number of physical hops* traveled by a message carrying a data item before either being discarded or delivered.

As the goal of the application developer is that of deciding *actions* based on data *sensed*, the first quantity intuitively measures the *quality of service* provided by the implemented system. The second measure indicates the cost paid to achieve a given degree of service, and is therefore key to understanding the *scalability* properties of the resulting implementation. The third measure gives more insights into the trends related to communication cost, describing



(a) Network overhead against number of highway sectors.

(b) Average number of physical hops traveled by a message.

Figure 12: Performance of the traffic management application.

where in the system communication takes place. As the independent variable, we choose to vary the number of highway sectors, as this dimension is likely to affect the system performance.

At a first glance, the aforementioned metrics appear to depend only on the performance provided by the run-time support. However, this is in turn affected by the particular task placement. Therefore, these quantities effectively provide insights into the effectiveness of the framework as a whole, from the ATaG compiler to the routing layer supporting Logical Neighborhoods. To compare against, we have chosen to compute the aforementioned metrics for an *ideal solution* minimizing the network overhead. This is determined by identifying the minimum cost routing tree connecting a sender to all the intended recipients, provided global knowledge of the network topology and reliable transmissions. The sender location is in turn determined by identifying, based on the same assumptions, the optimal task placement[3]. While being an artifact far from reality, this choice removes the bias introduced by comparing against alternative routing schemes that may not be expressly designed for the scenarios at hand. The performance obtained with a pure flooding scheme are also reported as an upper bound for further comparison.

Results. Given the message generation rates discussed earlier, our simulations revealed our solution can provide at least 96% of the actuations that would be occurring in case there were no message losses. This illustrates how the messages carrying the application data are effectively delivered to the intended recipients, and is consistent w.r.t the results shown in [18] obtained in a synthetic scenario. Remarkably, this metric is not affected by a varying number of highway sectors (and is hence not shown graphically). This behavior demonstrates how the processing is effectively kept in a limited portion of the system, both at the application and at the network level.

Figure 12(a) depicts the trends in network overhead against a varying number of highway sectors. As the chart illustrates, the network load imposed by our solution is much closer to the ideal solution than to flooding. More importantly, the pattern exhibited as the number of highway sectors increases mimics that of the ideal solution, while

[3]Determining the optimal task placement on a graph is a problem known to be NP-hard in the general case. To derive the optimal solution, we performed an exhaustive search in the space of all possible task allocations.

the flooding approach reveals a much steeper increase in the number of messages. These good scalability properties are clearly due to the ability of keeping message propagation localized around the nodes running the relevant tasks. Furthermore, albeit being already reasonable, these results are likely to see a dramatic improvement if the compiler is provided with a cost model of the underlying routing scheme, used to place the tasks smartly by minimizing a given metric. This topic is definitely worth being investigated, and is among our immediate research goals.

The chart in Figure 12(b) further confirms how the improvements over flooding are obtained by constraining message propagation around the nodes running related tasks. Indeed, the number of hops traveled by a message using flooding rapidly increases with the number of highway sectors. This is expected given the blind propagation of messages performed in this case. Differently, our solution keeps an almost constant performance in a range of settings, effectively ending up very close to the theoretical minimum. This trend demonstrates how the routing layer is well aware of the application semantics, that indeed requires a processing to span three adjacent highway sectors, and is therefore independent of the overall length of the highway.

8 Value of Scoping to Macroprogramming

Enabling scoping in macroprogramming makes developing complex applications extremely easy. At the same time, every programming model has its own specific field of applicability, and there is no "one size fits all" solution. Based on this, in this section we first evaluate *quantitatively* the programming effort in our reference application with respect to the total size of the deployed code, and then compare, on a *qualitative* basis, our programming model with existing solutions. The former gives a measure of how effective our approach is in automating the implementation process from high-level abstract specifications, hence alleviating the programming burden. The latter gives the bigger picture of the advantages brought by the combination of scoping and macroprogramming with respect to the current state of the art.

8.1 Evaluating the Programming Effort

To quantify the development effort, we took a number of code metrics on our prototype implementation[4]. Looking at the number of Java classes compiled to deploy the application on a single node, it turns out only 15 out of a total of 51 classes are the direct result of developers' effort. The remaining ones are either the implementation of the ATaG run-time support, or the Logical Neighborhood routing layer. 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. This is clearly due to the high-level abstractions provided by

[4] In doing so, we do not consider the code needed to implement the actual control algorithm, as it is strictly application dependent.

our framework, where most of the details related to message processing, coordination and communication are hidden from the application programmer.

Considering the code implementing each task, it is possible to identify a recurring pattern with only two classes needed. One of them is directly connected to the ATaG run-time, and contains processing that either inserts some data item in the the data pool (using `putData`), or handles the arrival of a new data item. Notably, in our implementation all the state variables defined in this class relate only to the application semantics, and never refer to distribution or coordination aspects. This same class usually holds a reference to a second class containing the actual processing, e.g., to average the incoming data as in the case of *AvgQueueLengthCalculator*. The data items are instead defined in separate classes. These usually implement only a number of setter/getter methods relative to different class attributes.

Notice how our framework naturally leads to highly encapsulated implementations: both within single tasks and with respect to different tasks. As for the former aspect, the data processing can be effectively implemented as an I/O machine without any explicit references to node locations or distribution. These low-level information can be encapsulated in the particular class connected to the ATaG run-time, where the references to the different tasks are implicit, being determined by the nature of data items and channel declarations. This results in highly re-usable implementations: adding an additional task or changing the system scale does not require any change in the application code.

8.2 Comparing against the State of the Art

Despite the clear advantages brought by the programming model we propose, scoping and macroprogramming might not be a suitable paradigm for every application. Here, we revisit the requirements illustrated in Section 2, and discuss the reasons why they cannot be addressed by existing programming models. Simultaneously, we highlight the requirements that *cannot* be effectively addressed by our proposal, and instead are better met by other frameworks.

For comparison, we will focus on the Regiment language [21], the Kairos system [11], and the programming model offered by Abstract Regions [26]. The former is a functional macroprogramming language based on the notion of region stream: a spatially distributed, time-varying collection of node states. These are taken as input to one or more functions used to express the application processing. Kairos is a macroprogramming model inspired by parallel architectures. Developers express the application behavior by writing or reading variables at nodes, iterating on the 1-hop neighbors of a node, and addressing arbitrary nodes. Abstract Regions is instead a node-centric programming approach, whose communication model —based on a data-sharing paradigm among nodes within a region— closely resembles the one in ATaG. In some cases, Abstract Regions is also able to select a subset of nodes in the system based on topological characteristics, thus also enabling a notion of scoping.

Multi-stage data processing. The combined use of *tasks* and *scopes* naturally allows the programmer to express

multiple stages of processing, and to determine the subset of nodes involved in each stage. Achieving the same in Kairos or Abstract Regions is more difficult, as neither of them embodies any well-defined notion of processing unit.

Conversely, Regiment is presumably even more effective than our framework in expressing this particular pattern. Indeed, as long as it is possible to express the input-output mapping as a mathematical function, composing multiple functions is straightforward in Regiment. The fundamental difference in this case is that our framework easily allows the output of a stage to be directed to more than a single, following stage. To the best of our knowledge, achieving the same in Regiment would force the system to duplicate the effort, invoking the same function more than once.

Multiple sub-goals. Similarly to what discussed above, scoping allows the definition of the different system partitions concerned with a specific sub-goal. This enables better separation of concerns, and results in more reusable implementations. It is hard to achieve the same in the absence of scoping, as in the case of Regiment or Kairos. In these cases, different programs should be written to achieve multiple goals without explicit support to achieve collaboration. In the case of Abstract Regions, one could, in principle, associate different goals to different regions, and run them in parallel. However, as any region requires an underlying, dedicated implementation reaching down to the network stack, the programming effort in this case may become unreasonable.

When the application has a single specific goal, our programming model can still be used to express the desired behavior, even if its expressive power is not fully exploited. However, if the developer needs to implement a *service* rather than an end-user application, e.g., a localization mechanism or a routing scheme, our framework might not be the best choice. Indeed, many details related to communication such as link quality are intentionally hidden from the programmer.

Localized interactions. The *channel annotations* we propose make it easy to describe the tasks involved in a given processing by placing a logical layer on top of the physical network. Regiment as well as Kairos are instead designed with a single system-wide processing in mind. Therefore it might be difficult to localize a given processing around specific nodes. In particular, the latter completely hides the individual devices from the programmer. This rules out the possibility of controlling how processing is distributed on the actual nodes.

However, if the desired goal depends on topological properties of the physical network, relying on Kairos or Abstract Regions might be advisable. In the former case, the network topology is explicitly made available with the construct to iterate on the 1-hop neighbors of a node. Therefore, the processing needed to, for instance, build overlays on top of the physical network can be expressed very succinctly [11]. In Abstract Regions, the network topology can be used as input during region construction. In this case as well, building overlay-like structures turns out to be easily achieved [26]. Conversely, our model might make expressing the aforementioned behaviors difficult if not impossible.

Addressing heterogeneity. To map a specific task to the nodes equipped with the required capabilities, we devised novel *instantiation rules*. Conversely, hiding the single nodes in Regiment requires all of them to produce the same

kind of data and have the same capabilities. The same reasoning holds in the case of Kairos: a node is characterized only by its identifier and the variables it exports to other nodes for read/write operations. In Abstract Regions, one might associate nodes with equal capabilities to the same region. However, as our reference application illustrates, nodes with the same characteristics do not necessarily communicate only among themselves. Therefore, one should address the problem of sharing data across different regions, a functionality not currently supported.

In case the system is mostly homogeneous, the drawbacks discussed above have no impact on the expressivity of the programming model. In particular, if the application is concerned only about the node identifier, the Kairos framework could provide a better fit between the high-level design and the implemented code, as it exposes the node identifiers as a first-class concept and provides built-in operators to manage them.

9 Conclusion and Future Work

Scoping gives developers the tools to address the complexity of large scale, sophisticated WSN applications. However, it is still relegated to node-level programming frameworks. In this paper, we introduced the notion of *scoping* in the context of *macroprogramming* sensor networks. Using macroprogramming, developers reason at a high-level of abstraction, where coordination and communication aspects are mostly hidden. To illustrate our proposal, we augmented the ATaG programming model with constructs enabling the definition of scopes. The feasibility of our approach is demonstrated by a dedicated compiler we developed targeting Logical Neighborhoods as supporting run-time, and by simulation studies assessing the performance of the resulting implementations.

In the near future we intend to explore techniques to *optimize the placement of tasks* on the nodes during the compilation process, looking at the expected flow of information as specified in the high-level abstract program.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

[10] Habitat Monitoring on the Great Duck Island. `www.greatisland.net`.

[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] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. of the* 33^{rd} *Int. Conf. on System Sciences*, 2000.

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

[14] W. Hu, C. T. Chou, S. Jha, and N. Bulusu. Deploying long-lived and cost-effective hybrid sensor networks. *Ad-Hoc Networks*, 4(6), 2006.

[15] Sun™ Java2 Micro-edition Specification, `java.sun.com/javame`.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Proc. of the* 3^{rd} *Int. Wkshp. on Sensor Networks and Systems for Pervasive Computing (PerSens - colocated with IEEE PERCOM)*, 2007.

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

[24] E. Petriu, N. Georganas, D. Petriu, D. Makrakis, and V. Groza. Sensor-based information appliances. *IEEE Instrumentation and Measurement Mag.*, 3:31–35, 2000.

[25] Sun™ Small Programmable Object Technology (Sun SPOT), `www.sunspotworld.com`.

[26] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc of the* 1^{st} *USENIX/ACM Symp. on Networked Systems Design and Implementation (NSDI)*, March 2004.

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