The Abstract Task Graph: A Methodology for Architecture-Independent Programming of Networked Sensor Systems*

Amol Bakshi*, Viktor K. Prasanna*
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089 USA
{*amol, prasanna*}*@usc.edu*

Jim Reich, Daniel Larner
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
{*jreich, larner*}*@parc.com*

Abstract

The Abstract Task Graph (ATaG) is a data driven programming model for end-to-end application development on networked sensor systems. An ATaG program is a system-level, architecture-independent specification of the application functionality. The application is modeled as a set of abstract tasks that represent types of information processing functions in the system, and a set of abstract data items that represent types of information exchanged between abstract tasks. Input and output relationships between abstract tasks and data items are explicitly indicated as channels. Each abstract task is associated with user-provided code that implements the actual information processing functions in the system. Appropriate numbers and types of tasks can then be instantiated at compile-time or run-time to match the actual hardware and network configuration, with each node incorporating the user-provided code, automatically generated glue code, and a runtime engine that manages all coordination and communication in the network. This paper primarily deals with the key concepts of ATaG and the program syntax and semantics. The end-to-end application development methodology is discussed briefly.

1 Introduction

Wireless sensor networks allow embedded, dense monitoring of the physical environment. The challenge in programming a sensor network is to coordinate the sensing, collaborative processing, and data flow in the network *correctly*, so that the desired functionality is achieved, and *efficiently*, such that performance requirements are met and the network lifetime is maximized. The need to manage a large collection of autonomous sensor nodes poses challenges from a programming perspective. State of the art programming languages and methods for sensor networks require the end user to manually translate the global application behavior in terms of local actions at each node, which is likely to be time-consuming and error prone for complex applications. Also, the application-level logic is tightly interfaced with the part of the program that coordinates lower level services such as resource management, routing, localization, etc. This lack of separation between system-level code and application-level code results in high complexity of coding non-trivial system behaviors.

There is a growing interest in *macroprogramming* of sensor networks [5, 10] which means moving beyond node-centric programming and instead specifying aggregate behaviors which are then automatically translated by a compilation framework into node-level specifications. This is motivated by the realization that the end user will be a domain expert and not a computer scientist, and will be primarily interested in the monitoring and control of physical phenomena. The details of in-network computing and communication which provides the desired functionality will be of incidental interest in most scenarios.

We introduce a macroprogramming model called the Abstract Task Graph (ATaG) that builds upon the core concepts of data driven computing and incorporates novel extensions for distributed sense-and-respond applications. The types of information processing functionalities in the system are modeled as a set of *abstract* tasks with well-defined input/output interfaces. User-provided code associated with each abstract task implements the actual processing in the system. An ATaG program is 'abstract' because the number and placement of tasks and the control and coordination mechanisms are determined at compile-time and/or run-time depending on the characteristics of the target deployment.

ATaG enables a methodology for architecture-independent development of networked sensing applications. Architecture independence is the ability to specify application behavior for a generic, parameterized network architecture. The same application may be auto-

*This work is supported in part by the National Science Foundation, USA under grant number IIS-0330445.

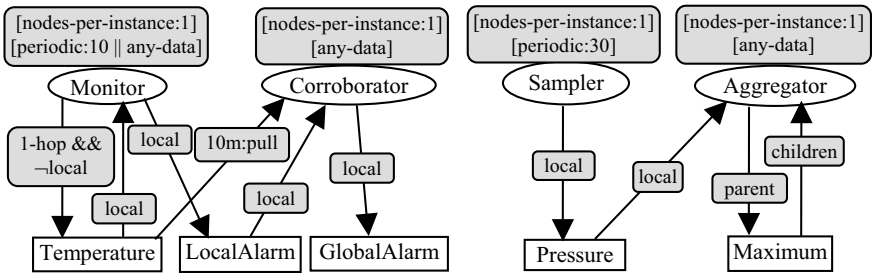

Figure 1: An ATaG program for environment monitoring

matically synthesized for different network deployments, or adapted as nodes fail or are added to the system. Furthermore, it allows development of the application to proceed prior to decisions being made about the final configuration of the nodes and network.

The focus of the ATaG approach is on simple specification of more complex patterns of information flow. A second objective is to define a process for automatically analyzing a user-supplied ATaG program and generating a deployment-specific distributed software system that consists of (a) the user-supplied application level code, and (b) a suitably customized runtime system responsible for control and coordination among the application level tasks. The present focus of our research is on defining the syntax and semantics of ATaG, and designing a runtime system and compiler for functionally correct translation of architecture-independent ATaG programs into architecture-specific node-level behaviors. Low level optimizations for a specific target platform have not yet been addressed.

Section 2 presents a sample ATaG program for an environment monitoring scenario with a view to highlight the key ideas before discussing them in more detail. Section 3 discusses the key concepts of ATaG and the syntax and semantics of ATaG programs. Details of the system level support for the ATaG model can be found in [1]. A brief overview of the end-to-end application development methodology is provided in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 An Illustrative Example

Figure 1 is an ATaG program for an environment monitoring system. The application is designed for a network of sensor nodes, each equipped with a temperature and a pressure sensor. The application exhibits two behaviors: the periodic computation and logging of the maximum pressure in the system, and the periodic monitoring of temperature. If the temperature gradient between a node and its neighbors exceeds a threshold, the node is required to corroborate the anomaly by surveying a larger area and then trigger an alarm. Corroboration helps to avoid false alarms due to a sensor malfunction.

The ATaG programmer first models each behavior in terms of a pattern of node-level interaction. In this case, the temperature monitoring requires a neighbor-to-neighbor exchange of temperature readings for gradient computation, and a many-to-one information flow for corroboration. The pressure monitoring and logging can be visualized in terms of information flow with incremental aggregation at each node of a virtual tree topology – a common pattern for efficient data aggregation.

The next step is to identify the types of processing and the types of data in the system, referred to in ATaG terminology as *abstract tasks* and *abstract data*. The abstract data for pressure averaging are: `Pressure` to represent the reading from the pressure sensor, and `Maximum` to represent the (partial) maximum. Similarly, the abstract data for temperature monitoring are: `Temperature` to represent readings from the temperature sensor, and `LocalAlarm` and `GlobalAlarm` to represent conditions where the threshold is violated and corroborated over a greater area, respectively. The abstract tasks for pressure monitoring are: `Sampler` to periodically record the pressure at the node, and `Aggregator` to track the maximum pressure readings from the node's children in the virtual tree. Similarly, the abstract tasks for temperature monitoring are: `Monitor` to compute local gradients, and `Corroborator` to analyze readings from the larger neighborhood. The programmer supplies the code for each abstract task and abstract data. This constitutes the imperative part of the ATaG program, and is also the *only code written by the programmer.*

The input/output interfaces of the abstract tasks are shown in the figure. Note that `Monitor` produces `Temperature` instances that represent local readings, and consumes `Temperature` instances produced by its neighbors. Since this distinction is between instances of data and not the type of data, the relationship between the abstract task and abstract data is both input and output.

The final step is to associate annotations (depicted by shaded, rounded rectangles) to indicate task placement and information flow patterns. For instance, the annotations indicate that `Monitor` is to be instantiated on

every node in the system, run periodically, and also executed when a new `Temperature` instance is available. The `Temperature` instance produced by `Monitor` is to transmitted to a11 1-hop neighbors and not added to the local data pool. `Corroborator` will be triggered by the production of a `LocalAlarm` by `Monitor`, 'pull' all instances of `Temperature` within a 10 meter radius, and possibly produce a `GlobalAlarm`. For the other behavior, the `Aggregator` is to be executed whenever an instance of `Pressure` is produced locally (by the periodic `Sampler`) or an instance of `Maximum` is received from any of the node's children. The output of `Aggregator` is to be transmitted up the virtual tree maintained by the runtime.

3 Programming with ATaG

3.1 Key concepts

ATaG is based on two key concepts: data driven program flow and mixed imperative-declarative specification.

In *data driven computing*, tasks are passive objects that are defined in terms of their input and output interface. Tasks do not interact with each other. The basic primitives available to the programmer are `getData()` and `putData()` for consumption and production of data items respectively from the *data pool*. A task is automatically scheduled for execution when its operands become available. This scheduling is performed by an underlying runtime system that manages the data pool. The data driven paradigm is attractive for several reasons. Tasks can use data items at the desired level of abstraction without worrying about how they are produced. Programs are highly extensible and reusable because there is no direct task-to-task coupling. From an implementation perspective, data driven programming can be naturally supported by an event driven runtime system, resulting in efficient resource utilization. An 'event' is the production or consumption of a data item from the data pool.

Mixed imperative-declarative specification facilitates a clear separation of functionality from other non-functional aspects such as task placement and coordination. For sensor networks, this separation is especially critical because it allows the same program to be synthesized without modification onto various deployments by interpreting the declarative part differently. Also, we chose to design a visual programming interface for specifying the declarative aspect, thereby eliminating the need for programmers to learn a new syntax. Support for both network awareness and network transparency through such separation of concerns has been explored in the distributed computing community [6, 8] – our motivation is to design suitable mechanisms that are useful for networked sensing applications.

3.2 Syntax

ATaG captures global application behavior in a *network-aware* but *network-independent* way. The close coupling of sensor networks with the physical environment, and the dependence of in-network processing on the spatio-temporal location of data being processed mandate network awareness. On the other hand, requirements of portability and architecture independence require a model and representation that is network independent. Note that ATaG does not hide parallelism. The compiler translates a concise and architecture-independent but explicitly parallel specification into the node-level control and coordination behavior.

An ATaG program is a set of *abstract declarations*. An abstract declaration can be one of three types: *abstract task*, *abstract data*, or *abstract channel*. Hereafter, we occasionally omit the word 'abstract' for sake of brevity when the meaning is apparent. Each abstract declaration consists of a set of *annotations*. Each annotation is a 2-tuple where the first element is the *type* of annotation, and the second element is the *value*.

Abstract task: Each abstract task declaration represents a *type* of processing that could occur in the application. The number of instances of the abstract task existing in the system at a given time is determined in the context of a specific network description by the annotations associated with that declaration. Each task is labeled with a unique name by the programmer. Associated with each task declaration is an executable specification in a traditional programming language that is supported by the target platform. Table 1 describes the annotations that can be associated with a task declaration in the current version of ATaG.

Abstract data: Each abstract data declaration represents a *type* of application-specific data object that could be exchanged between abstract tasks. ATaG does not associate any semantics with the data declaration. The number of instances of a particular type of data object in the system at a given time is determined by the associated annotations in the context of a specific deployment and depends on the instantiation and firing rules of tasks producing or consuming the data objects.

Each data declaration is labeled with a unique *name*. Similar to the executable code associated with the task declaration, an application-specific *payload* is associated with the data declaration. This payload typically consists of a set of variables in the programming language supported by the target platform. No annotations are currently associated with abstract data items.

Abstract channel: The abstract channel associates a task declaration with a data declaration and represents not just which data objects are produced and/or consumed by a given task, but which instances of those types of data

Type: Instantiation	
value[:parameter]	**Description**
one-on-node-ID:*id*	Create one instance of the task on node *id*
one-anywhere	Create one instance of the task on any node in the network
nodes-per-instance:[/]*n*	Create one instance of the task for each *n* nodes of the network. When *n* is preceded by a "/", create exactly *n* instances of the task and divide the total number of nodes into *n* non-overlapping domains, each owned by one instance.
area-per-instance:[/]*area*	Same as for nodes-per-instance. Parameter denotes area of deployment instead of number of nodes. The non-overlapping domains are in terms of area of deployment, not number of nodes.
spatial-extent:$x_1, y_1, x_2, y_2, \ldots$	Create one instance of the task on every node that is deployed in the polygon defined by the co-ordinates (x_1, y_1), (x_2, y_2), ..., (x_1, y_1).

Type: Firing rule	
value[:parameter]	**Description**
periodic:*p*	Schedule task for periodic execution with period of *p* seconds.
any-data	Schedule task for execution when at least one of the input data items are available.
all-data	Schedule task for execution only when all the input data items are available.

Table 1: Abstract Task: Annotations

Type: Initiation	
value	**Description**
push	The runtime system at the site of production of each instance of the associated abstract data item is responsible for sending the instance to nodes hosting suitable instances of the consumer task(s).
pull	The runtime system at the node hosting an instance of the consumer task is responsible for requesting the required instance(s) of the associated abstract data item from the site(s) of production.

Type: Interest	
value[:parameter]	**Description**
[¬]local	Channel applies to the local data pool of the task instance. The negation qualifier excludes the local data pool, and can be used in conjunction with other qualifiers.
neighborhood-hops:*n*	Channel includes all nodes within the *n*-hop neighborhood of the node hosting the task instance
neighborhood-distance:*d*	Channel includes all nodes within a distance *d* of the node hosting the task instance
all-nodes	Channel includes all nodes in the system
domain	Channel includes all nodes that are owned by the task instance. This value is used in conjunction with the nodes-per-instance or area-per-instance values of the Instantiation annotation of the abstract task.
parent	Channel applies to the parent of the node hosting the task instance - in the virtual tree topology imposed on the network by the runtime system.
children	Channel applies to all children of the node hosting the task instance - in the virtual tree topology imposed on the network by the runtime system.

Table 2: Abstract Channel: Annotations

items are of interest to a particular instance of the task. Table 2 describes the annotations that can be associated with an abstract channel in the current version of ATaG. The abstract channel is the key to concise, flexible, and architecture-independent specification of common patterns of information flows in the network. For instance, spatial dissemination and collection patterns may be expressed using simple annotations such as "l-hop," "local," or "all nodes," on output and input channels. More sophisticated annotations may be defined as needed or desired for a particular application domain.

3.3 Semantics

The two basic primitives available to the programmer are `getData()` and `putData()` for consuming and producing data items. The runtime system manages the *data pool* and moves data between producers and consumers. We now briefly summarize the semantics of ATaG.

If the task is *periodic*, it is scheduled for execution when the periodic timer expires, regardless of the state of its input data items. The per-task timer is set to zero each time the task begins execution and is said to expire when the timer value becomes equal to the task's period. If the task is *any-data*, it is scheduled for execution as soon as a new instance of any of its input data items is available. If the task is *all-data*, it is scheduled for execution as soon as a new instance of each of its input data items is available. Other valid firing rules are *periodic* $\vee$ *any-data* and *periodic* $\vee$ *all-data*.

Each well-behaved task must invoke exactly one `getData()` call for each of its input data items, and may invoke at most one `putData()` for each of its output data items. `getData()` is a destructive read from the task's perspective. Once a particular instance of a data item is read by a task, it is considered to be eliminated from the data pool as far as that task is concerned.

Task execution is atomic. Each application-level task will run to completion before another application-level task can begin execution. All members of the set of dependent tasks of a particular data item are executed before other tasks that might be dependent on the output data items of the tasks in this set are executed. Whenever the production of an instance of a data item results in one or more of its dependent tasks to become ready, those tasks will consume the same instance when they invoke a `getData()` on the input data item. This means that the particular instance that triggered the task(s) will not be overwritten or removed from the data pool before every scheduled dependent task finishes execution. The implication of this is that `putData()` is not guaranteed to succeed if an instance of the abstract data item being produced cannot be overwritten. Such situations might arise if, for example, the rate of production of an abstract data item is greater than the rate of consumption. The application developer is responsible for checking for success or failure of `putData()`.

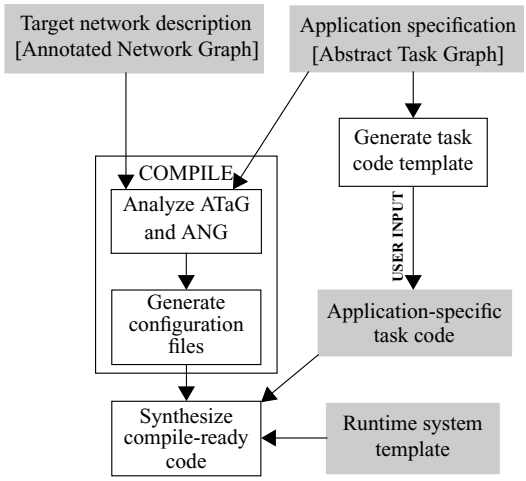

Figure 2: Application development with ATaG

4 Application Development Methodology

Figure 2 depicts the application development methodology using ATaG. The application developer graphically inputs the declarative part of the ATaG program and a description of the target deployment in the form of an annotated network graph (ANG), which is not discussed in this paper. The ANG contains information such as the number of nodes, the co-ordinates of each node, network connectivity, etc. A code generator analyzes the ATaG program, determines the I/O dependencies between tasks and data objects, and generates code templates for the abstract tasks and data. The programmer populates the code templates with application functionality. The compiler then interprets the program annotations in the context of the ANG, and generates configuration files for each node that customize the behavior of that node based on its role in the system. Finally, compile-ready code is generated for each node in the network.

The graphical interface to the programming and synthesis environment is through a configurable graphical tool suite called the Generic Modeling Environment (GME) [4]. The declarative part of the ATaG program which consists of the various declarations and their annotations is specified visually. In fact, we exploit the composability of ATaG and allow users to create libraries of ATaG programs that can be simply concatenated to build larger applications. GME stores the model defined by the user in a canonical format. Tools called *model interpreters* can read from and write to this model database. In our case, model interpreters were written for the components represented by unshaded boxes in Fig. 2.

5 Related Work

The core ideas of ATaG have been applied in different contexts in the distributed computing community. Modular, extensible, and convenient parallel programming was the motivation for the data driven paradigm in the Data Driven Graph [11] model. Separating the core application functionality from other concerns such as task placement and coordination was one of the primary motivations of efforts such as Distributed Oz [6] and IBM's PIMA project [2]. Tuple spaces [3, 9] provides a content addressable persistent shared memory for coordination across distributed processes. Although the notion of a shared data store also exists in ATaG, our 'active' data pool abstraction that schedules tasks based on the availability of data instances is fundamentally different from the 'passive' tuple space whose modifications are really a side effect of task execution on different nodes.

TinyDB [7], Regiment [10], Kairos [5], and Semantic Streams [12] are examples of macroprogramming methodologies for sensor networks. While ATaG explores a mixed imperative-declarative programming style and data-driven program flow, TinyDB provides a declarative SQL-like query interface for sensor data. Regiment is a demand-driven functional language based on Haskell, with support for region-based aggregation, filtering, and function mapping. Kairos is an imperative, control-driven programming paradigm that provides a distributed shared memory abstraction to the node level program. The Semantic Streams markup and declarative query language, based on Prolog, is used to specify queries over semantic information directly, while the actual selection, wiring, and optimization of low level modules to implement the querying is performed by a service composition framework.

6 Conclusion and Future Work

We have presented a novel method for specifying sensor network programs based on the Abstract Task Graph. ATaG programs consist of two parts: a declarative section, which specifies the connectivity between tasks and constraints on their placement and communication; and an imperative section, with the implementation of the tasks written in a traditional computer language. In the current system, these tasks are instantiated on the nodes at compile-time, but in future work, we plan to investigate fully dynamic versions, instantiating tasks based on the current network hardware and connectivity and the underlying measurements. This will probably be restricted to the more capable hardware platforms. In any case, the declarative specification is fully portable to both other network architectures and other hardware architectures, requiring at most a porting of the individual task code, which represents only a small fraction of the system functionality.

This paper focused on the key concepts of ATaG and the syntax and semantics of ATaG programs. We did not cover issues such as the operational semantics of the compiler and synthesis system, valid and invalid constructs in ATaG and opportunities for performance optimization in the runtime. Finally, the set of annotations and the synthesis and analysis tools available thus far are limited; for the platform to achieve wider applicability, we expect to extend these significantly in the future.

References

[1] A. Bakshi, A. Pathak, and V. K. Prasanna. System-level support for macroprogramming of networked sensing applications. In *Intl. Conf. on Pervasive Systems and Computing (PSC)*, June 2005.

[2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An application model for pervasive computing. In *6th Annual ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, 2000.

[3] C. Curino, M. Giani, M. Giorgetta, A. Giusti, G. P. Picco, and A. L. Murphy. Tiny Lime: Bridging mobile and sensor networks through middleware. In *3rd IEEE Intl Conf on Pervasive Computing and Communications*, 2005.

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

[5] R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming wireless sensor networks using Kairos. In *Intl. Conf. Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[6] S. Haridi, P. V. Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[7] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *2nd Intl. Workshop on Information Processing in Sensor Networks*, 2003.

[8] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *21st Intl. Conf. Software Engineering*, 1999.

[9] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *PerCom*, March 2004.

[10] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *1st Intl. Workshop on Data Management for Sensor Networks (DMSN)*, 2004.

[11] V. D. Tran, L. Hluchy, and G. T. Nguyen. Data driven graph: A parallel program model for scheduling. In *Proc. 12th Intl. Workshop on languages and Compilers for Parallel Computing*, pages 494–497, 1999.

[12] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, April 2005.