Srijan: A Graphical Toolkit for WSN Application Development

Animesh Pathak and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA 90089
{*animesh, prasanna*}*@usc.edu*

Abstract

Macroprogramming is an application development technique for wireless sensor networks (WSNs) where the developer specifies the behavior of the system*, as opposed to that of the* constituent nodes*. Although several languages for macroprogramming have been proposed recently, the area of easy-to-use graphical tools for macroprogramming in real environments is still largely unexplored.*

In this paper, we present Srijan*, a toolkit that enables application development for WSNs in a graphical manner using data-driven macroprogramming. It can be used in various stages of application development,* viz. *i) specification of application as a task graph using a graphical interface, ii) customization of the auto-generated source files with domain-specific imperative code, iii) specification of the target system structure, iv) compilation of the macroprogram into individual customized runtimes for each constituent node of the target system, and finally v) deployment of the auto generated node-level code in an over-the-air manner to the nodes in the target system.*

The current implementation of Srijan *targets the recently released Sun SPOT sensor nodes, and is available for public download. Our experiments show that using toolkit can drastically reduce the time and effort involved in developing real-world WSN applications.*

1. Introduction

Sensor network macroprogramming aims to aid the wide adoption of networked sensing by providing the domain expert the ability to specify their applications at a high level of abstraction. This is in contrast to the initial days of wireless sensor networks (WSNs), when application developers had to manually customize a set of node-level protocols to achieve system-wide goals, thus making the process difficult for the *domain experts*, who were not well-versed in the intricacies of distributed computing.

Since the goal of WSN macroprogramming research is to make application development easier for the *domain expert*, we believe that it is absolutely necessary to make *easy-to-use toolkits* for macroprogramming available to them in order to both make their task easier, as well as to gain feedback about the macroprogramming paradigms themselves. Although various efforts exist in literature for making WSN application development easier (see Section 5), very few general purpose graphical toolkits are publicly available for the application developer to choose from. We believe that toolkits supporting alternative paradigms will greatly aid the application developers, who will have a wide-range of programming styles to choose from, depending on application, as well as personal stylistic choice.

In this paper, we present the design and implementation of *Srijan* – a graphical toolkit for WSN application development (named after the Sanskrit word for creation). Using it, the developer can create the applications using the data-driven ATaG [1] macroprogramming language. Section 2 provides a brief overview of ATaG, and introduces the building heating, ventilation and air-conditioning (HVAC) application we use in this paper as an illustrative example.

Contributions. The main contribution of this paper is an easy-to-use graphical toolkit for reducing the burden of each stage of WSN application development. In Section 3, we provide details of the design of our toolkit, which consists of the following components:

- **Task Graph Specification GUI** - using which the developer can specify the details of the ATaG

This work is partially supported by the National Science Foundation, USA, under grant number CCF-0430061 and CNS-0627028.

task graph representing the application.

- **Customizable Code Auto-generator** - which generates templates of imperative code that the developer can then fill-in.
- **Target System Description GUI** - for loading and editing the description of the target WSN in a graphical manner.
- **Compilation and Deployment Module** - for creating customized node-specific code, building node-level binaries, and deploying them onto the constituent nodes in the system.

We have also ported the DART runtime system of ATaG to the Sun SPOT [2] sensor nodes, a recent release from Sun that is rapidly emerging as a popular platform for WSN application developers. We have modified the ATaG compiler so that it produces task templates in J2ME, as opposed to the J2SE code generated by the earlier version. Additionally, we have also made the *sensors and actuators* of the Sun SPOT nodes available to the *Srijan* users via a set of easy-to-use libraries. By making *Srijan* a free download, we intend to target the above community, as well as system programmers who wish to contribute to our runtime system libraries.

We would like to note at the outset that the focus of this paper is on the toolkit and the graphical user-interface provided by it, and *not* on the compilation of macroprograms. Compilation of ATaG programs, and the optimizations involved in the process, are discussed elsewhere (e.g., in [3]). *Srijan* can easily be modified to support any enhancements in the ATaG compiler itself.

In Section 4, we show results from our work in developing two realistic applications – building environment management (HVAC) and highway traffic management. Our experiments show that using *Srijan*, application developers can specify and deploy their applications in a timely fashion, while having to write $\sim 2\%$ of total system code (or $< 10\%$ of application-specific code). Section 6 concludes with our plans for future work.

2. Background

To provide the reader the necessary background, we summarize the salient features of ATaG in this section. Readers familiar with the programming style of ATaG can skip to Section 2.2, where we discuss an application that we will use as an illustrative example in this paper.

2.1. ATaG Macroprogramming Framework

The Abstract Task Graph [1] (ATaG) macroprogramming framework consists of an extensible, high-level programming model, a corresponding node-level runtime system, and a dedicated compilation framework to generate node-level code. An ATaG program is written in a *data-driven* manner using a mixed *imperative-declarative* programming model. The declarative portion of an ATaG program – a task graph – consists of the following components (see Figure 1 for details).

- Abstract Data Items: The main currency of information in an ATaG program. They represent the information in its various stages of processing inside a WSN.
- Abstract Tasks: These represent the processing performed on the abstract data items in the system. Tasks do not share state with other tasks, and can communicate only by producing and consuming data items. Tasks are annotated with *instantiation rules*, specifying where they can be located, as well as *firing rules*, specifying whether a task is triggered periodically or due to the production of certain data item(s).
- Abstract Channels: These connect tasks to the data items consumed or produced by them, and are annotated with logical scopes [4], which express the *interest* of a task in a data item.

The above task graph is complemented by *imperative code* for each data item and task. The developer uses this code to specify the processing that occurs when a task fires. Note that due to the data-driven programming model provided by ATaG, this imperative code does not have any inter-task communication function calls other than consuming or producing data items (using the `handleDataItemProduced()` and `putData()` primitives respectively).

The Data-driven ATaG Runtime (DART) [5] provides the necessary abstraction of a distributed data-pool for ATaG programs while hiding the underlying, platform-specific details. 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. The *DataPool* is responsible for managing all instances of abstract data items produced or consumed at the node. The *LogicalNeighborhoods* module handles data delivery by implementing a dedicated routing scheme. Finally, the *NetworkStack* is in charge of communication with other nodes in the network, and manages the physical layer protocols.

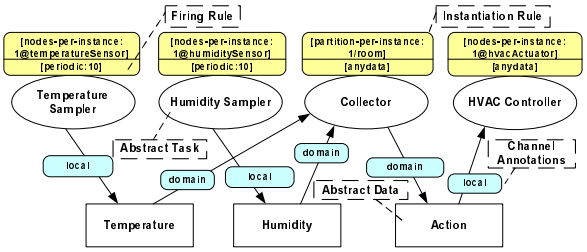

Figure 1. An ATaG program for building environment management.

The *input* to the *ATaG Compiler* [3] consists of the ATaG task graph, and the imperative code for each task and data item. In addition, the details of the target system, including the node locations and list of attached sensors and actuators, is also provided. The compiler then decides the placement of the tasks of the individual nodes. The *output* of this process is deployable code for each node, consisting of the tasks assigned to it. Additionally, the compiler generates *customized DART modules* for each node, containing the logical scopes where the data produced at the node is to be sent.

2.2. Reference Application

In [6], the authors described how ATaG's data-driven macroprogramming approach can be used to express a wide range of WSN application behavior, including **a)** *Hierarchical Data Gathering* – similar to the initial data-gathering WSN applications, **b)** *Localized Interactions* – where a set of nodes correlate their sensed data to make a decision, and **c)** *Actuation Driven by Sensing* – where the data sensed by nodes equipped with sensors is used to trigger the actions of other nodes equipped with actuators in a heterogeneous system.

For illustrating the ease-of-use of developing applications in *Srijan* we focus on the following building environment management application (HVAC), similar in spirit to other applications in the literature [7]. We consider a set of nodes spread across a building, with each node possibly attached to a temperature sensor, a humidity sensor and an actuator that can control the temperature and humidity of a region. The aim of the system is to maintain desirable temperature and humidity levels in each room of the building, by correlating the information from the sensor installed in the room, and using it to drive actuation.

Figure 1 describes our application as an ATaG task graph. To address the heterogeneous nodes in the system, we define the placeholders *temperatureSensor*, *humiditySensor*, and *hvacActuator* according to

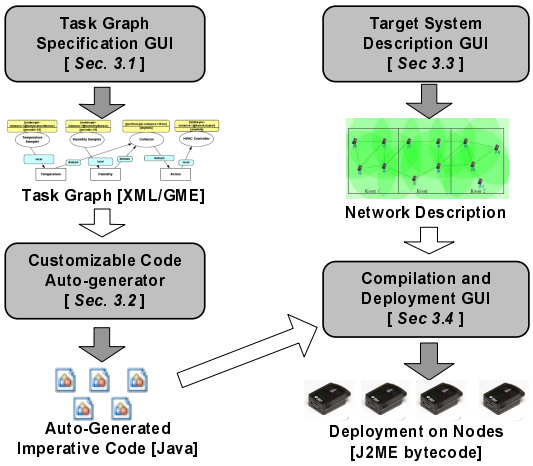

Figure 2. Overview of Application Development using *Srijan*

the sensors and actuators that may be attached to the system's nodes. The *Temperature Sampler* and *Humidity Sampler* tasks – instantiated on the corresponding node type – sample their surroundings and generate the *Temperature* and *Humidity* data items periodically (see their `periodic:10` firing rules). To process the data produced by them, we placed a *Collector* task in each room. To express this, we use the `partition-per-instance:1/room` instantiation rule. This partitioning can be used as a domain, and binds the *Collector* task to the *Humidity* and *Temperature* data items produced on the same floor. Upon processing the data, if needed, the *Collector* produces a command for the actuating tasks in the form of an *Action* data item. The *HVAC Controller* task is placed on all nodes of type `hvacActuator` and responds to the *Action* data item by adjusting the temperature/humidity controls.

3. Components of the Toolkit

Figure 2 shows the various components of our toolkit that the application developer can use. The clear arrows show the *inputs*, while the gray arrows show the *output* of each component. The various components of *Srijan* are as follows.

3.1. Task Graph Description GUI

The ability of specifying a WSN application in a graphical manner as interconnected task and data items is a major part of ease-of-use provided by ATaG. In *Srijan*, we have customized the Generic Modeling Environment (GME) [8] for providing this facility to the application developer. Figure 3 shows the application

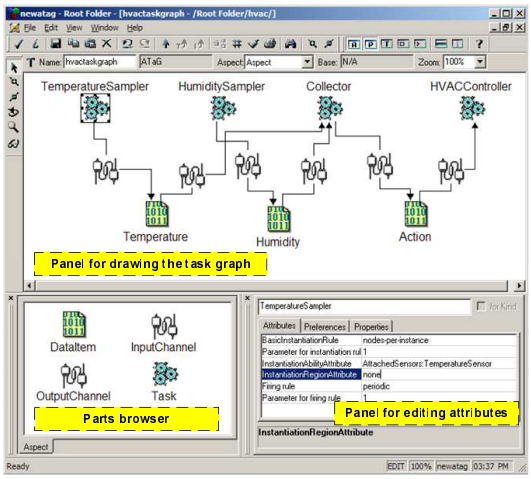

Figure 3. Building Environment Management (HVAC) application in our task description GUI

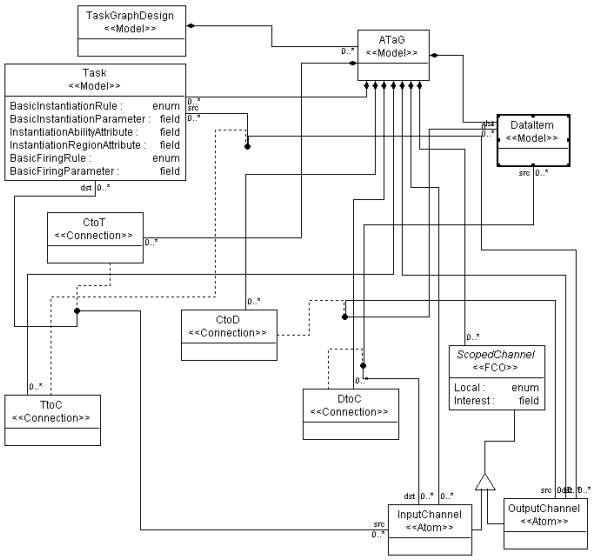

Figure 4. GME MetaModel for ATaG

introduced in Section 2.2 specified using our GUI. The developer can drag and drop *parts* representing the tasks, data items, and channels of the application from the *parts browser* onto the workspace, and draw connections between them to show their inter-relationships. The annotations of each component can then be set by clicking on it and editing the attributes in the *attribute editing panel*. Once the details of the task graph are specified, *Srijan* generates an XML file representing it by invoking the *PatternProcessor* model interpretor of GME. This XML representation of the task graph can then be used by the other components of our toolkit, discussed later in this section.

Our work towards providing the above facilities to the programmer consisted of two parts. Firstly, we developed a *metamodel* in GME for ATaG, (Figure 4) describing the possible attributes of each component of an ATaG program, as well as the relationships between them. Secondly, we developed a *pattern file* for ATaG, which is be used by the GME pattern processor to generate a properly formatted XML file given a particular ATaG task graph.

3.2. Customizable Code Auto-generator

As stated previously in Section 2, an ATaG program consists of two parts - the *task graph* representing the properties and inter-relationships of the abstract tasks and data items, and *imperative code* expressing the details of each task and data item. This component takes the XML file generated by the GME pattern processor and generates the following files:

- **IDConstants.java**: This contains the declarations mapping each task and data item's ID to a static variable to enhance readability.
- **[DataName].java**: For each data item, *Srijan* creates a file with the needed Java headers, and methods for serialization and deserialization of the data item. The file also contains dummy code for the contents of the data item itself, with guidelines on how to edit it.
- **[TaskName].java**: For each abstract task, *Srijan* creates a file which describes the task as a Java class. For periodic tasks, the code for ensuring that the task fires with the desired frequency is pre-built into the auto-generated code. The user only needs to edit the `handleExpiryOfTimer()` method. For tasks that are fired due to the presence of data item(s), the auto-generated code contains sample code for methods for handling the data item produced.
  In both cases, this component generates sample code for calls to the `putData()` method of the runtime system, using which the developers can make the tasks interact with the DataPool.
- **PreBuiltAtagManager.java** This file contains the code for determining the logical scope corresponding to each data item, so that the runtime can deliver it accordingly as it is produced. Note that this file does not need any editing from the application developer.

The code generated at this stage by *Srijan* is compatible with the widely used NetBeans™Java IDE, and can be easily accessed in it for editing and syntax check. Figure 5 shows the file auto-generated by *Srijan* for the *Collector* task of the HVAC application. Note that the developer only has to insert

```
/** Collector.java */
...
public class Collector implements Runnable {
        ...
        public void run() {
                /*Checking for each DataItem, to see which one was produced*/
                DataItem t_dataItem;
                if((t_dataItem = m_dataPool.getData(IDConstants.T_COLLECTOR,
                        IDConstants.D_TEMPERATURE)) != null){
                        Temperature recvdTemperature =
                            (Temperature) t_dataItem.core();
                        this.handleTemperatureProduced(recvdTemperature);
                }else if((t_dataItem =
                           ...
                }
        }

        private void handleTemperatureProduced(Temperature r_Temperature) {
       /* Sample code - please write your own */
       /* You can also use the putData Code Samples below */
                int var1 = r_Temperature.getvar1();
                m_myState.debugPrint("[Collector]Got a Var1 " + var1);
        }
        private void handleHumidityProduced(Humidity r_Humidity) {
            ...
        }
    /* Sample code for producing dataItem Action
        Action m_Action = new Action();
        m_Action.setvar1(this.m_myState.getMyId());
        m_dataPool.putData(new DataItem(IDConstants.D_ACTION,
        IDConstants.T_COLLECTOR, m_Action));
    */
}
```

Figure 5. Auto-generated code for the *Collector* task

code in the `handleHumidityProduced()` and `handleTemperatureProduced()` methods. *Srijan* also provides code samples to copy-paste for generating the *Action* data item produced by the *Collector*.

3.3. Target System Description GUI

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

The information about the nodes in the target system can either be uploaded into *Srijan* en-masse as a text file, or by using the GUI to add individual nodes and editing their attributes such as position, network address, and sensor and actuators attached.

The top part of Figure 6 shows the components of our toolkit used for specifying the target system structure. The GUI consists of two main components, a display screen at the left side of the main window and a tabbed control panel which contains three sub-panels - *Editor*, *Viewer*, and *Compiler*.

The editor and viewer panel aim to provide facilities to give the application developer an intuitive view of a system while configuring it. The properties of the nodes can be both seen and edited using the tool. In addition, the various visualizations (network links, radio ranges, sensing range) allow the developer to see the connectivity and coverage of the target system. These visualizations can also be used to show a subset of the information, such as radio links in a specific partition/group or between nodes with a specific type of sensor etc.

3.4. Compilation and Deployment Module

The lower half of Figure 6 depicts the module of our toolkit that provides the application developer the ability to tune the compilation parameters and deploy the generated code to the nodes in the target system. After the developer has specified the path to the root directory of the target code, he can compile the ATaG program to individual node-level files. Since the current ATaG compiler supports only the *random* option for optimization, *Srijan* allows the setting of the randomization seed. However, the toolkit can easily be extended to support more optimizations as they are developed.

For deploying the code on the Sun SPOTs, our toolkit uses a Sun SPOT base station node for upload-

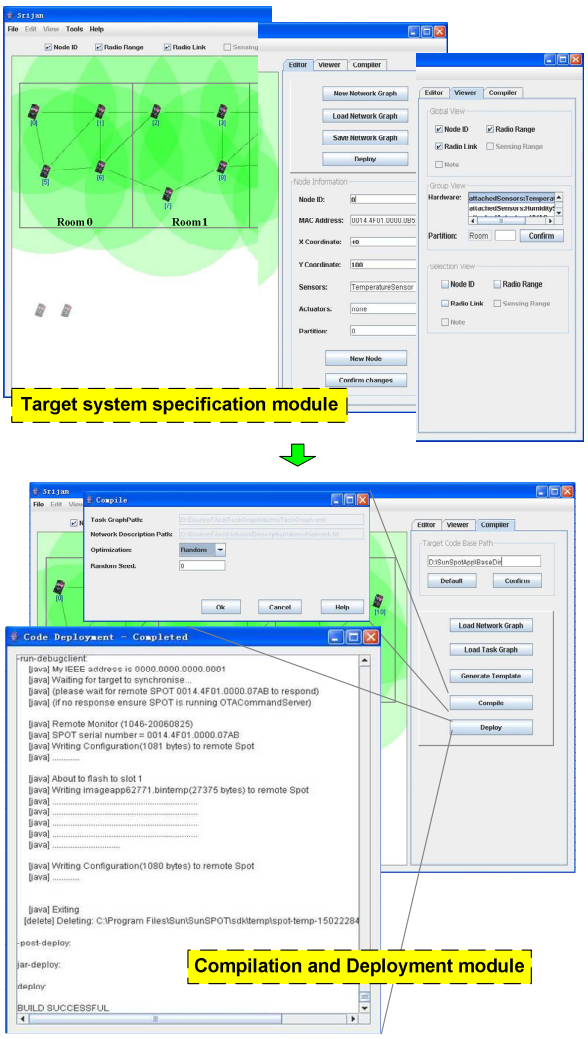

Figure 6. Network description, compilation and deployment using *Srijan*

ing it to the nodes via OTA(over the air) commands. This phase uses the information about the MAC addresses of the individual Sun SPOTs specified in the earlier stage.

4. Evaluation

To evaluate the performance of *Srijan*, we developed a set of applications on it, including the HVAC application, which we have used as our illustrative examples throughout this paper, and the highway traffic management application used in [4]. For each of the applications, we performed the complete end-to-end development – starting from specifying the ATaG task graph to deployment of code on the nodes – using *Srijan*. We used a Pentium-4 2.8 GHz laptop with 1GB of RAM running Windows XP for our evaluation. The deployment was done onto the Sun SPOT [2] nodes, with a 180 MHz 32 bit ARM920T processor, 512K RAM and 4M Flash memory. The nodes run the Squawk Java virtual machine directly out of flash memory, and can run programs written using J2ME libraries. The Sun SPOT base station was used to deploy the code over-the-air (OTA) to the SPOTs. We used the Java hProf profiler for measuring execution time.

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

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

The data from our experiments is summarized in Table 7. Note that the time taken by *Srijan* to generate the files are within acceptable limits, and are limited only by the hardware it is being run on, and in the case of deployment, also on the Java compiler used by the Sun SPOT SDK. More importantly, the developer had to write only a very small fraction of Java source files. The *total code* deployed on each node consists of three components – **a)** Base Template Code - containing the DART libraries, **b)** Application-specific Auto-generated Code - generated by *Srijan*, and **c)** User-generated Code - written by the application developer to specify the details of the task and data. Figure 8 shows that the user-generated code is only around 2% of the total code. Even if we neglect the library code, *Srijan* generated $> 90\%$ of the application-specific code in each case. The importance of the time taken by

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

Figure 7. Costs involved in various stages of application development using *Srijan*

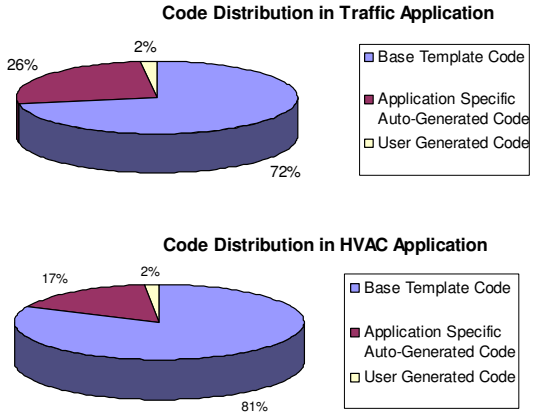

Figure 8. Distribution of coding effort

the application developer in specifying the task graph and customizing the auto-generated code is highlighted by the fact that under normal circumstances, *Srijan* will be used by domain experts, e.g. civil engineers, who would have taken much more time customizing the runtime protocols and figuring out the task placements if it was not available as part of Srijan. We believe that our experiments demonstrate that our toolkit makes application development for WSNs more convenient for the domain expert, and we look forward to feedback from developers who download and use our toolkit.

5. Related Work

Despite many years of research in the area, application development for WSNs is still largely done by writing *text-based code* for *individual nodes*, be it nesC on the Mica motes or C on the BTNodes. One of the earliest toolkits proposed to reduce the programming effort was the Sensor Network Application Construction Kit (SNACK) [9], which provides a component composition environment that allows developers to define explicit configurable parameters for application-level components. The SNACK user develops applications at the *node-level* using a text-based description of *wiring* between components, several of which are libraries provided by the authors. These programs are analyzed by the compiler to generate maximally-shared nesC expansions, which then have to be deployed just like normal nesC applications. The Flask language [10] facilitates *node-level* programming using data-flow graphs and provide facilities for composing atomic subgraphs across the network using a *flow* communication model. The application is specified in a variant of OCaml, and the behavior of individual processing elements is specifies in nesC. The Flask compiler then generates node-level nesC code from the datagraph. This approach of mixed imperative-declarative programming is very similar to ATaG, but Flask currently allows application description only at the node-level. On a different direction of research, the Deployment Support Network WSN development toolkit [11] aims to aid the programmer by collecting data about the deployed sensor network in an over-the-air fashion. In addition to the above, some graphical toolkits have also been proposed for WSN application development. The authors of Viptos [12] allow developers to model and simulate TinyOS applications in a graphical manner. A similar functionality is provided by GRATIS [13] where developers can use GME for easy modeling of TinyOS applications. However, in both these tools, the developer still has to reason at the *node level*, while *Srijan* is geared for enabling developers to think at a much higher level of abstraction.

At a *system-level* of abstraction, one stream of research has focused on treating the WSN as a database, and making it easy for developers to write and deploy data-querying applications on sensor networks. The Task [14] toolkit makes designing and deploying TinyDB query-based application easy, where users can query the sensor data using SQL-like queries, and

also provides a visualizer for monitoring the network health and sensor readings. Semantic streams [15] presents each user with a 3D rendering of the sensors in the testbed as well as all predicates that are queryable. [16] builds on it by providing a spreadsheet approach to programming and managing data-querying applications in WSNs. In semantic middleware [17], applications are represented in a graphical interface as composable data sources and inference units which can be connected to retrieve required data by composition engines. jWebDust [18] provides a multi-tier application environment, where different sensor networks can be visualized as one to query the sensed data in a user-friendly manner. While these toolkits help WSN developers by allowing them to reason at a high-level, the developer can specify the application as a query-based system only.

Another domain-specific *system-level* project is EnviroSuite [19], which is targeted at tracking applications. In EnviroSuite, an application contains a list of *objects*, specified in a textual manner, which are abstractions of environment elements. Their toolkit provides keywords and method libraries to define objects and hide the fact that the execution of object methods may need distributed computing across network from programmers. The target code is in nesC, and is deployed in the usual fashion. In the field of environment monitoring, [20] presents a user friendly toolkit, where application developers can specify their application in an Eclipse-based GUI, as well as properties of the target network. The compiler-generated code must then be deployed manually to the target nodes in the system. In contrast, *Srijan* is a more general-purpose tool, which can be used to design and deploy a variety of data-driven WSN applications.

Programs in the general-purpose Regiment [21] macroprogramming language are written in a functional programming style, and are compiled by the compiler into an intermediate representation called the token machine language (TML). nesC implementation of the same is currently underway. *Srijan*, on the other hand, provides an easy-to-use graphical interface for data-driven macroprogramming and compiles to node-level Java code. Perhaps the closest tool to our work is VisualRDK [22], which enables the developers of pervasive applications to easily develop applications for heterogeneous systems using a graphical toolkit, where individual tasks can communicate using simple triggers. *Srijan* provides a similar, easy-to-use graphical interface specifically geared towards developing complex applications for sensor networks using data-driven macroprogramming.

6. Concluding Remarks

In this paper, we introduced *Srijan* – a graphical toolkit for end-to-end development of WSN applications using the ATaG data-driven macroprogramming language. The toolkit is aimed to facilitate domain-experts in all stages of application development, namely task graph specification, imperative code customization, specification of target system structure, compilation of the macroprogram to customized node-level code, and deployment of machine level code to the constituent WSN nodes. We believe that by providing a graphical interface to all the above steps, *Srijan* will help bringing WSNs to a broader community of users. Our experiments show that using *Srijan*, developers can quickly develop realistic WSN applications while writing a very small fraction of the actual application code.

Although *Srijan* helps make WSN application design easy, it is up to the designers of the ATaG compiler and DART runtime system to make up for the loss of performance (e.g., high energy costs, shorter lifetimes) that inevitably accompany a rise in the level of abstraction. The current implementation of the ATaG compiler is more a *macro-expanding task allocator* than a true compiler, since it does not perform any optimizations while determining task placements and customizing the runtime system. Two such optimizations that can be looked at are placing tasks to reduce to communication costs and increase the system lifetime; and performing logical-expression sharing when combining the scopes in the channel-annotations of the ATaG task graph.

Since *Srijan*'s aim is to provide easy-of-use to the WSN application developer, future work on it will focus mostly on the interface provided to the application developer, as well as the libraries available on the WSN-platform specific code. The current version of our toolkit is written mostly in Java, and produces code to be deployed on the Sun SPOT nodes. We have publicly released the toolkit [23], as well as its source code, so that while application developers can start using it for their goals, system developers can freely port it to another platform, such as ConTiki on the motes, Linux on GumStix and the StarGate, C on the BTNode etc. Our other future work on *Srijan* is in several concurrent directions. We are working towards integrating the SWANS/Jist simulator [24] with the toolkit, so application developers can debug and profile their code before deployment. In the network description GUI, we are planning to incorporate support for 3-D deployments, as well as letting the application developer specify situations where certain nodes, although physically proximate, can not communicate (perhaps

due to the presence of a concrete wall between them). Most importantly, we will look forward to feedback from the users of *Srijan*, since making their work easy is what the system aims to achieve.

References

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

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

[3] A. Pathak, L. Mottola, A. Bakshi, G. P. Picco, and V. K. Prasanna, "A compilation framework for macroprogramming networked sensors," in *Proc. of the the 3^{rd} Int. Conf. on Distributed Computing on Sensor Systems (DCOSS)*, 2007.

[4] L. Mottola, A. Pathak, A. Bakshi, V. K. Prasanna, and G. P. Picco, "Enabling Scoping in Sensor Network Macroprogramming," in *Fourth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS) (accepted)*, 2007.

[5] 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.

[6] 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.

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

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

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

[10] G. Mainland, M. Welsh, and G. Morrisett, "Flask: A language for data-driven sensor network programs," in *Technical Report TR-13-06, Harvard University Technical Report*, 2006.

[11] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum, "Deployment support network - a toolkit for the development of WSNs," in *European Workshop on Wireless Sensor Networks (EWSN)*, 2007.

[12] E. Cheong, E. A. Lee, and Y. Zhao, "Joint modeling and design of wireless networks and sensor node software," Electrical Engineering and Computer Sciences University of California at Berkeley, Tech. Rep., 2006.

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

[14] "Tiny application sensor kit," http://berkeley.intel-research.net/task/.

[15] K. Whitehouse, F. Zhao, , and J. Liu, "Semantic streams: A framework for composable semantic interpretation of sensor data," in *European Workshop on Wireless Sensor Networks (EWSN)*, 2006.

[16] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao, "A spreadsheet approach to programming and managing sensor networks," in *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, New York, NY, USA, 2006.

[17] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye, "A semantics-based middleware for utilizing heterogeneous sensor networks," in *Proc. of the 3^{rd} Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2007.

[18] I. Chatzigiannakis, G. Mylonas, and S. E. Nikoletseas, "jWebDust : A java-based generic application environment for wireless sensor networks," in *DCOSS*, 2005, pp. 376–386.

[19] 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.*, vol. 5, no. 3, 2006.

[20] J.-P. Arp and B. G. Nickerson, "A user friendly toolkit for building robust environmental sensor networks," in *CNSR '07: Proceedings of the Fifth Annual Conference on Communication Networks and Services Research*, 2007, pp. 76–84.

[21] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proc. of the Int. Conf. on Information Processing in Sensor Network*, 2007.

[22] T. Weis, M. Knoll, A. Ulbrich, G. Mhl, and A. Brndle, "Rapid prototyping for pervasive applications," *IEEE Pervasive Computing*, vol. 6, no. 2, pp. 76–84, 2007.

[23] "*Srijan* - graphical WSN application development toolkit," http://srijan.googlepages.com/.

[24] R. Barr, Z. J. Haas, and R. van Renesse, "Jist: an efficient approach to simulation using virtual machines," *Softw. Pract. Exper.*, vol. 35, no. 6, 2005.