Fall 2009

Programming and self stabilization for wireless sensor networks

Kajari Ghosh Dastidar
University of Iowa

Recommended Citation

Ghosh Dastidar, Kajari. "Programming and self stabilization for wireless sensor networks." PhD (Doctor of Philosophy) thesis, University of Iowa, 2009.
http://ir.uiowa.edu/etd/363.

PROGRAMMING AND SELF STABILIZATION FOR WIRELESS SENSOR NETWORKS

by

Kajari Ghosh Dastidar

An Abstract

Of a thesis submitted in partial fulfillment
of the requirements for the Doctor of
Philosophy degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Professor Ted Herman

ABSTRACT

Ubiquitous computing has become a widespread phenomenon in today's modern world, with the computing technology integrating with our daily life in an invisible manner. *Embedded systems* and *wireless sensor networks* are popular choices to achieve this. Programming embedded and sensor network systems has always been a challenge for the programmers due to the lack of sufficient *high-level* programming support. To deal with this serious limitation, we have developed DESAL (Dynamic Embedded Sensing and Actuation Language) which is a user-friendly high-level programming language for wireless sensor networks with an integrated *middleware*, which hides the *low-level* detail from the programmers. In this thesis we present the design and development of DESAL.

We have made DESAL programs *rule based*. Programs are written in guard-action format defined in terms of the program states. There are established formal correctness proving methods that can work on guard-action formats to mathematically check a program for errors. Also, there is no hidden control context like events or interrupts. *Time synchronization* has been developed as part of the middleware that lets DESAL programs to coordinate through synchronized actions throughout the network. This facilitates classic coordination algorithms like clock synchronization, spanning tree construction and consensus. Also, synchronized wake up saves energy. *Neighborhood management,* including node discovery and monitoring, is also provided by the middleware. DESAL programs communicate via *state sharing*. There is no network programming required. The middleware provides that automatically. Combining all these features DESAL provides major network management services, and yet presents the users with a simple high-level programming interface. We implemented the DESAL compiler to convert DESAL programs to NesC on TinyOS and to Java.

Another novel feature we have introduced in DESAL is a variable of type '*token'*. The concept of token is commonly used in mutual exclusion algorithms. One of the case studies we have done uses the token variable to achieve increased lifetime of sensors in a ring topology. The working of token is hidden from the user. Another case study with tokens involves selective activation of RFID tags in a scenario where among the three RFID tags present only one can work at a time.

Struct is a new data structure introduced in DESAL. Sometimes we need to group together two or more variables. It is important to receive them at the same time. Hence, it is important to send these grouped data over the radio together. Struct does that.

Function is another newly added feature to DESAL. Function is added to group together repeated statements in a program. The unique feature of function is that, it uses only global variables. No new local variable is declared. This can significantly reduce the stack overhead of the program, thus saving memory and running time.

Case studies have been done to illustrate the features of DESAL and to find scope for improvement.

Abstract Approved: ______________________________
Thesis Supervisor

____
Title and Department

____
Date

PROGRAMMING AND SELF STABILIZATION FOR WIRELESS SENSOR NETWORKS

by

Kajari Ghosh Dastidar

A thesis submitted in partial fulfillment
of the requirements for the Doctor of
Philosophy degree in Computer Science
in the Graduate College of
The University of Iowa

December 2009

Thesis Supervisor: Professor Ted Herman

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Kajari Ghosh Dastidar

has been approved by the Examining Committee
for the thesis requirement for the Doctor of Philosophy
degree in Computer Science at the December 2009 graduation.

Thesis Committee: ______________________________
Ted Herman, Thesis Supervisor

____
Sukumar Ghosh

____
Steve Bruell

____
EJ Jung

____
Geb Thomas

ACKNOWLEDGMENTS

I would like to acknowledge my Ph.D. advisor Prof. Ted Herman for his help at every step of my study and his guidance and support. I would also like to thank Prof. Sukumar Ghosh for his moral support and kind advice throughout my stay in this university. Thanks are also due to my other Ph.D. committee members Prof. Steve Bruell, Assistant Prof. EJ Jung and Prof. Geb Thomas for their valuable inputs in shaping my thesis. I would like to acknowledge the emotional support and constant encouragement given to me by my family.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

This dissertation presents a new programming language called DESAL (Dynamic Embedded Sensing and Actuation Language) for wireless sensor networks. DESAL is a high-level programming language with an integrated middleware hiding low-level detail from the user. The usual target audience using the sensor networks includes hydrologists, civil engineers and doctors. They are expected to program the sensor networks to customize them to cater to their own needs. But at present the widely used programming platforms are suited to expert programmers who can master low-level programming primitives, operating system components and network protocols. This makes programming very difficult for consumers. This is where DESAL comes into play. DESAL provides the user with a friendly high-level programming interface where there is no need to write the codes for low-level operating systems and network protocols. In this chapter we discuss the motivation behind the development of the language, our contribution, and the organization of the subsequent chapters.

1.1 Embedded Systems And Sensor Networks

Pervasive computing has become a widespread phenomenon in today's modern world with the computing technology integrating with our daily life in an invisible manner. In most of our modern amenities, ranging from leisure technologies like cell phones, remote controls, home appliances to emergency life-saving technologies, we have underlying functioning computers. In the early years of the 20th century, *Embedded Systems* technology came into being to meet the demands of invisible integration of computing services to the day-to-day amenities.

Embedded systems are constituted of specialized and application-specific computing systems usually integrated on small microchips, which are embedded in the electronic devices. Programming embedded systems has always been a challenge for the embedded systems engineers and the computer science programmers due to the lack of sufficient high-level programming support. For each *microprocessor*, an application developer is expected to learn the system specifications given in the long detailed datasheets, know the intricacies of the circuit designs, bus protocols, and many more details. With the rapid advance in the field of embedded technology, we can presently boast of having thousands of different embedded systems catering to the modern human civilization in every sphere of life. At the same time, with the thousand different technologies, it becomes necessary for an application developer to gain detail knowledge of each of the different systems. To deal with this serious limitation, researchers have engaged in developing general-purpose software support for programming embedded systems.

Considerable research work is going on in the area of the development of high-level software support for embedded systems. The aim is to provide user-friendly software support to operate and manage the embedded device. By user-friendly we mean the software should hide the low-level hardware detail from the programmer, making application programming simpler. In addition to that, such software should ideally support various hardware platforms, so that there is no need for a programmer to learn different software for programming different systems. Real-time operating systems like VxWorks, WinCE, PalmOS, QNX, etc were developed to provide users with an execution system similar to a desktop system. These are still popular for PDAs, cell phones, set-top boxes, etc. Though these operating systems are not the best in terms of minimal memory usage, they provide adaptive microkernel, multiprogramming support, reliability and memory protection. Smaller operating systems like Creem, pOSEK, Ariel, etc. were developed for more application specific purposes. But, ground breaking research work in the hardware technologies have resulted in the development of smaller embedded systems demanding more resource efficient operating systems and the supporting software tools [5].

A significant breakthrough in the embedded world was the development of *sensor devices*. These are special types of devices, which can interact with its environment by sensing temperature, pressure, movement, etc. A sensor device comprises of sensors integrated on small chips, which can be programmed for various sensing applications like, setting up the fire alarm in case of fire, reporting abnormal temperature or pressure fluctuations, etc. For sensing over a wide area, sensor devices (termed as sensor node or mote in a network) are deployed in a network forming a Wired Sensor Network (also known as *Sensor-Actuator Network*). Such applications include, determining if a room is empty, detecting uniform heating of a room, etc. The sensor nodes are usually very cheap and small in size enabling deployment in a large number for more area coverage and accurate readings.

The wired technology soon became a hindrance to the distribution of large number of sensor nodes, as it was very difficult and clumsy to set up a wired network just anywhere, especially outdoors. As an answer to this, *wireless sensor network technology* was developed. A wireless sensor network is comprised of multiple sensor motes, which can be deployed virtually anywhere on earth, including physically inaccessible places. The sensor nodes form an *ad-hoc network*, which can be monitored remotely.

Research in sensor technology have seen major development in the hardware technologies, resulting in smaller and cheaper sensor nodes. This made possible the deployment of a large number of nodes in a network to cover a widespread area and get a huge amount of sensor readings for accurate calculation. The dense deployment has resulted in high precision of the data collected by these motes. But, at the same time, the sensor nodes have become significantly resource limited (in terms of memory, battery life and processing power). With increasing demand of sensor networks, applications have become more complex, and consequently it is becoming more and more challenging to program these resource poor sensor motes to support the advanced applications. The current operating systems and the programming languages are too much application-specific and low-level to support such dynamic and advanced programming. In an attempt to solve this problem, researchers are developing new operating systems and high-level middleware abstractions to make it easier for the application developers to write more complex and dynamic programs.

1.2 Our Contribution

Our contribution includes development of a high-level user friendly programming language called DESAL (Dynamic Embedded Sensing and Actuation Language), which addresses the above-mentioned challenges. This language hides the low-level details from the programmer making programming simple and therefore, less error prone. DESAL has an integrated middleware constituting of the low-level services like time synchronization, message communication, neighborhood management and dynamic binding. These services are automatically provided to the user. Thus, the user is not required to write low-level codes for operating systems and network protocols. This work is jointly done with Dalton and Hallstrom of Clemson University. We have taken the language grammar from Clemson, and created our own compiler in *DParser for Python*. We have integrated our compiler with Clemson's Java program to convert a DESAL program to equivalent NesC code. Our work also includes writing codes in Python to convert a DESAL program to an equivalent Java program. The Java program can communicate with the sensor nodes via the SerialForwarder. One problem with directly using the serial port is that only one PC program can interact with the mote. Additionally, it requires one to run the application on the PC, which is physically connected to the mote. The SerialForwarder is a graphical tool, which can remove both of these limitations. More than one program running on the PC can send packets to the

SerialForwarder, which is displayed in its interface. This tool can connect to the attached mote via the serial port. [54]. A novel contribution to DESAL is the addition of *token* type variables. Tokens allow the users to achieve mutual exclusion in the network. We have shown two case studies to illustrate the usage of tokens. *Structs* and *functions* are also newly added features to DESAL. Structs allow grouping together two or more variables. Functions group together repeated statements. The DESAL approach cannot do everything. It has some overhead. It cannot react in microseconds to events. It is not in the style of programming language like TinyOS, which is event-based.

1.3 Outline Of Thesis

The thesis is organized as follows: In chapter 2 we discuss some related work in a chronological manner. In chapter 3 we talk about our language DESAL. In this chapter we discuss the underlying middleware features, the runtime architecture, and look at a sample DESAL program along with the language grammar. We have shown how the high-level programming interface makes writing program in DESAL simpler than that in NesC. It has been validated by comparing the Blink program written in NesC with that written in DESAL. We have added new language features to DESAL. They are structs, functions and tokens. The new inclusions are validated with case studies. In chapter 4 we present token type variables. In this chapter the usage of token is illustrated in detail with the help of two case studies. Structs and functions are discussed in Chapter 5 along with case studies. Chapter 5 also explains how DESAL is converted to Java. A DESAL program can compile to NesC and Java. The conversion to NesC has been done in collaboration with Dalton and Hallstrom of Clemson University. The files needed to convert DESAL programs to Java are written in Python. An example of a converted Java program from a DESAL program is given in the Appendix A. The program given in Appendix A has been run in the lab where the Java program communicated with attached base station via the SerialForwarder. We conclude the thesis in chapter 6 followed by suggested future work. The numbers given in brackets throughout the thesis refer to the papers having same numbers in the bibliography.

CHAPTER 2
LITERATURE REVIEW

The release of TinyOS and NesC in the year 2000 was the first significant step towards the development of a *real time* operating system and programming language for sensor networks. But, major limitation of this system is that, in order to maintain hard real time requirements, the programmers are required to do low-level component wiring. This makes complex application development difficult. TinyOS is very much event-driven in its style of programming. The event-driven nature and the lack of some context memory (like a stack for separate threads) necessitate a kind of "call back" style. In TinyOS this is called "*split control*". For example, you call X.start(), and then later you receive an X.startDone() event. You cannot just call X.start() and expect the program to wait until the X.start() finishes. This style can be very confusing to non-experts. Also, TinyOS/NesC allows only static wiring, thus making *dynamic programming* impossible. To overcome these limitations research have been done in developing various advanced programming support. The major research works done in this area can be categorized based on the different programming principles adopted. This has been illustrated in Figure 1. This categorization is built on the survey done by Hadim and Mohamed [39]. We have also reviewed a few additional papers for categorization. Literature review has been done in order to be aware of the past and present related research work and in that context understand the unique contribution of DESAL. In chapter 3 we will discuss DESAL in detail and show why it is different than the research work done before it. The different research work has been presented in a chronological order. We have shown how the research in this area has developed over time, what has been accomplished, and what needs to be done. In the following section we give a brief overview of the work done by the different researcher. In the next section we will look into some of the major papers in more detail.

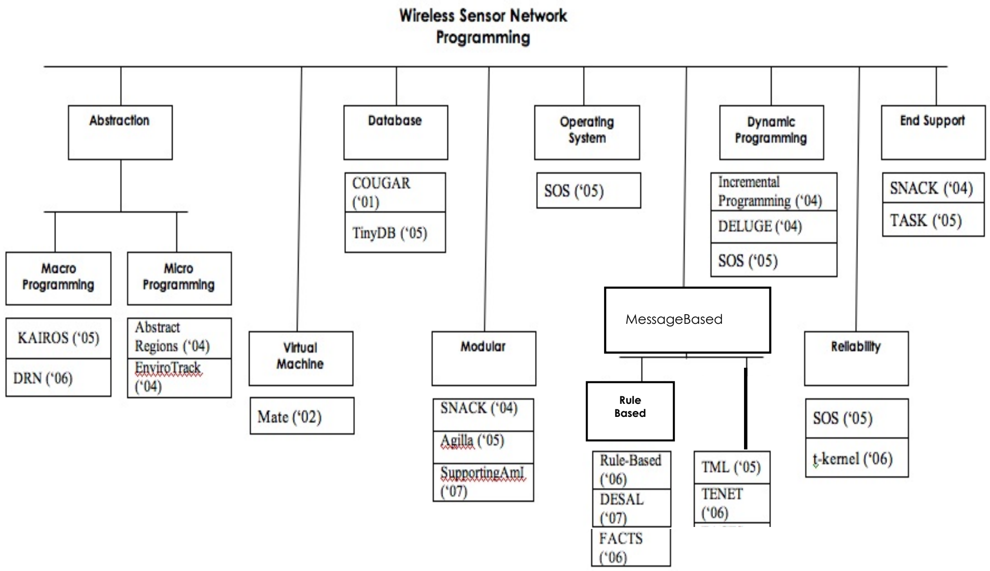


Figure 1 This tree attempts to categorize the various research done in different areas of wireless sensor network programming .The different research papers have been referred to by short names, which are mentioned below with references. Against each research paper the year of publication is put in parenthesis.

2.1 Brief Overview

The following section gives a brief categorized overview of the papers listed in the category tree in Figure 1, followed by detailed discussion of some of the most relevant papers.

To program in TinyOS/NesC, knowledge of low level detail is needed. The first natural response to this drawback was the development of an efficient middleware on the top of existing platform. In 2002, Mate [8] was released. Mate is a tiny communication-centric virtual machine built on top of TinyOS, which provides simpler programming interface and supports dynamic reprogramming of the network. Mate was followed by multiple projects in the development of dynamic reprogramming techniques.

In 2004, Incremental Programming technique [14] and Deluge [10] was introduced. Both of these support frequent reprogramming of the sensor motes already deployed in large networks.

In the same year (2004), Sensor Network Application Construction Kit (SNACK) [16] was developed in UCLA. This was a successful attempt towards the implementation of a new configuration language, component and service library, and compiler, making sensor network programming simpler and more efficient compared to TinyOS/NesC alone. Agilla [19], designed around the same time (2005), provided a higher level programming interface through mobile agent and tuple-space abstraction.

During the same time (2004), microprogramming (local) techniques like Abstract Regions [15] and EnviroTrack [13] were developed with a goal to provide a simpler higher-level interface to the programmers. Microprogramming means writing lowest machine-level code for programming microchips.

The next year (2005) saw the implementation of a new operating system altogether, named as SOS [21]. A significant improvement over TinyOS was that SOS supported dynamic memory allocation resulting in network reprogramming along with offering reliability. Also, compared to Mate, it provided better higher level programming abstraction with more flexibility and less CPU overhead.

Macroprogramming refers to high-level programming. Unlike microprogramming, in this case, a high-level programming language is used for coding, which hides all the low-level (machine-level) detail from the programmer. Macroprogramming languages, like Kairos [20] was developed in the same year (2005). Kairos provides a global view of the system to the programmer, hiding the low-level detail, making programming simpler. Development of another macro programming concept, Declarative Resource Naming (DRN) [35] followed closely in 2006.

In the same year (2005), the embedded research group at Berkeley released TinyDB [17], a SQL based database management system tailored for the sensor network system. In 2001, another sensor network database system, named COUGAR [7], was developed. But TinyDB was a much-improved version, which instantly became popular because of its familiar SQL interface and compatibility with TinyOS.

The same group then implemented TASK (Tiny Application Sensor Kit) [24] on TinyDB, providing a very user-friendly interface for non-programmers to deploy and manage sensor network applications.

In 2005, an intermediate rule (embedded in tokens)-based language, Token Machine Language (TML) [18], was developed, which can be targeted by compilers for higher-level systems. A rule is a condition based on the state variables of the system and/or events. TML provides a layer of abstraction for lower-level runtime environment, such as TinyOS. Abstraction relieves the programmer from dealing with the low-level events, making programming simpler and less error-prone. The year 2006 saw more emphasis on the rule-based language development. Program codes in TENET [28] are deployed encapsulated in tasks. This also helps in code reusability increasing programming efficiency. Also, the main contribution of TENET was that it presented a tired architecture to increase the system manageability.

Another middleware, FACTS [29], provided a rule-based language with an underlying event-based architecture making programming simpler while attempting to minimize resource utilization. A rule is a condition based on the state variables of the system and/or events. Action(s) against a rule gets executed when the rule evaluates to true. In FACTS, an action can change the state of the system, as well as trigger events. This makes FACTS a rule-based as well as an event-driven architecture.

But the event-driven rule-based languages have a limitation. They allow external events to determine scheduling of the rules. So, the programmers cannot be totally oblivious to the event management. The Rule-Based Language [34] was proposed as an improvement to the previous work. In addition to being a rule-based language, it is also state-based, so that the programmers are required only to deal with the system state, and not worry about the events. Also, since the scheduling of rules is predictable in this case, it is possible to determine the program correctness during compilation, thus increasing the reliability of the program.

2.2 Review In Detail

The present need in the field of sensor network programming is to have a high-level programming language to make the life easier for the programmers while attempting to minimize the limited resource utilization. Since the development of TinyOS and NesC, several research efforts have been done in an attempt to design more and more sophisticated sensor network programming support. Some researchers have focused on the development of high-level middleware, while some have designed intermediate high-level languages. Development of operating systems and macro and micro programming abstraction concepts also got significant attention. Some tools have been developed in the recent years to provide a high level interface to the non-programmers for managing the sensor networks. This section of the report attempts to give a broad overview of the current state of the art in the development of high-level programming languages. Figure 2 shows a timeline diagram of the significant developments in the area of sensor network programming.

2.2.1 Abstract Regions [15]

Abstract Regions is an abstraction over the local functionality providing flexible control over resource consumption improving communication and data accuracy and at the same time simplifying application development by hiding the low-level detail from the programmer.

Abstract Regions programming provides a local area communication abstraction to the application developer. For a node, the abstract region comprises of the neighbors of the node. The abstraction hides the details of low-level communication among the neighbors of the nodes, and provides data aggregation and compression related services transparently. The abstraction provides a simple programming interface while local data processing increases the accuracy of the processed data, and significantly reduces the communication overhead of the entire network saving energy.

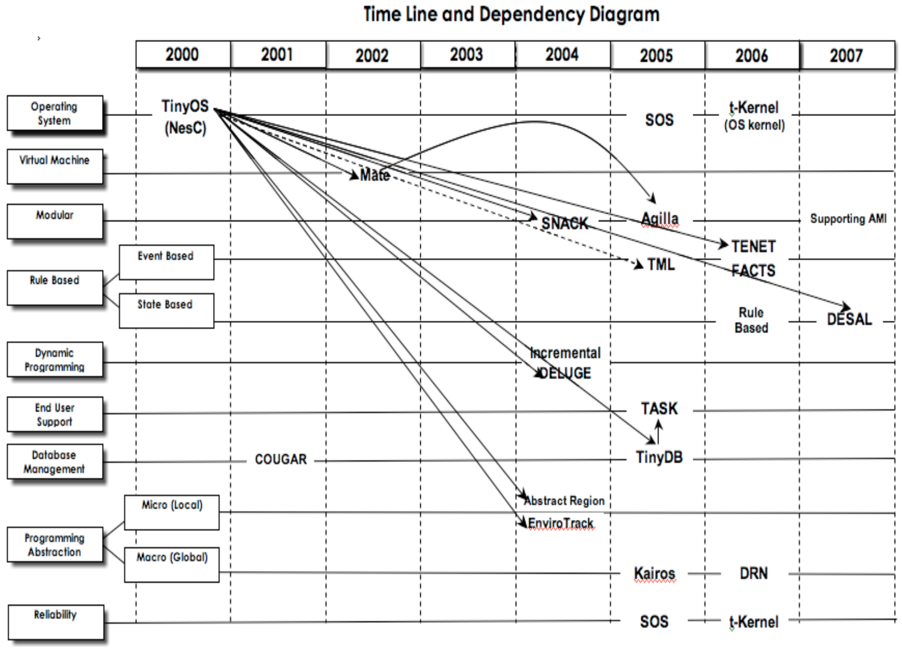


Figure 2 This is a chronological diagram showing the major research works done in the past seven years, since the release of TinyOS/NesC. The arrows illustrate the platform dependence of the research. The dotted arrow signifies that the application can also support other platforms.

The definition of neighbor is application-specific and is defined in the program as ‘neighborhood relationship’. A set of implementations of regions with their associated data reduction algorithms (N-radio hop, k-nearest neighbor, etc) has already been provided, which the programmers can simply call in their program as a function.

Abstract region implements a blocking, synchronous interface using fibers. This interface makes application development simpler as there is no need to write different event handlers for a program. A fiber is a thread-like abstraction for an execution instance added to TinyOS. The enhanced TinyOS has a default system fiber, which is non-blocking and event-driven. The other fibers are for the application, which can be blocked. An application fiber also has the option to be event-driven. Thus, a simple programming interface is presented retaining the goodness of the event-driven model.

The abstract region supplies the programmer with four operators, hiding the implementation details:

- Neighbor discovery: creates a set of neighbors based on the region definition. There is an option to update the set periodically.
- Enumeration: returns the set of active neighbors in the region in order to access them.
- Data Sharing: allows data sharing among the neighbors through the get (retrieve a remote variable value) and put (store the remote value to a local variable) functions.
- Reduction: applies user specified data aggregation and compression algorithms on the shared variables.

A tuning interface is provided to fine-tuning resource consumption based on the 'quality measure feedback' generated by the above operator functions. This improves energy usage and reliability of the network.

Abstract Regions programming model has been successfully implemented and deployed on the TinyOS platform. The authors analyzed the model based on four applications. The analysis satisfactorily validates the claim. More regions need to be implemented to enrich the programming architecture. Implementation of tools for the programmers to understand resource usage and quality tradeoff can enable them to have more control over the fine-tuning.

2.2.2 TinyDB [17]

TinyDB simplifies data-driven application development by hiding the low-level data management detail and providing the simple well-known SQL interface for data queries. TinyDB is a data query processing system for wireless sensor networks. The TinyDB architecture provides the sensor network user with a database file called ‘sensor’, which stores the sensor values collected from the network. The user queries ‘sensor’ using a user friendly SQL-like interface via a base-station. TinyDB uses resource-aware algorithms to collect and manage (aggregation, filtering, etc) sensor data and hides the associated low-level programming details from the users.

TinyDB is a distributed query processor running on each node in the network. Users submit their queries to the network via a base-station. The queries are parsed and power optimized in the base-station before they are disseminated to the network through a power aware routing tree. The processed results are routed back to the base via the same tree, and the users are presented with the query results, which gets stored in the ‘sensor’ table.

Acquisitional Query Processing (ACQP) is adopted for TinyDB along with the traditional query features provided by SQL. ACQP determines when, where and in what order the data is to be collected to minimize power usage of the network and at the same time providing increased data accuracy, compared to the traditional SQL query processing methods. The ‘sensor’ table, thus, gets updated only when a query is generated to save resources, and at the same time presenting up-to-date values to the user.

A user can specify the sample period of a query as a query parameter. Many queries, like, event-based, grouped aggregation and actuation queries, provides useful services to sensor network applications. TinyDB also supports data logging and network health monitoring through special queries. Queries can be prioritized.

TinyDB has been successfully implemented and deployed. At present, it is the most efficient and widely used query processing system for the wireless sensor networks. Sophisticated prioritization methods need to be designed to improve the real-time response.

2.2.3 TML [18]

Token Machine Language (TML) provides a framework for network coordination and hides the low-level detail from the programmers making sophisticated application development simpler, while economizing resource utilization.

TML has been designed as an intermediate language on an abstract machine called Distributed Token Machine (DTM), providing a layer of abstraction on the underlying event-driven operating system, such as TinyOS. Tokens constitute the unit of computation and communication in the sensor network deploying DTM/TML. Tokens are small messages through which data and programs get communicated to the sensor nodes. A token handler is invoked upon the receipt of a token, which executes the token. Execution of a token atomically changes the state of the token, and the state of the system. DTM hides the low-level distributed execution and communication detail from the programmers. The TML compiler provides memory protection. A TML program, based on object oriented programming model, simply needs the programmer to write codes for tokens without worrying about their scheduling and coordination across the nodes, which are taken care of by the DTM. This provides a simple and expressive high-level abstraction to the application developers, while taking care of resource utilization.

DTM manages the tokens and their coordinated computation and communication in the network by specifying token handler, token scheduling and their storage (dynamic memory allocation for new tokens) in each node. Currently, TML/DTM supports only TinyOS/NesC. DTM is presented as a TinyOS module. Handlers are compiled into corresponding NesC commands. A token is disseminated across the network over communication channel by encapsulating in a TinyOS Active Message.

The scheduling algorithm chosen is usually implementation-specific. Prioritized scheduling (fast response to events) and static scheduling (improved runtime) is supported. The DTM specifies handler language, which is currently a simple C-based restricted language (TML). DTM provides reliable communication by taking care of message loss, and providing some fault-tolerance.

DTM provides a collection of gradient network coordination operations through a gradient interface added to TML. Gradients can provide efficient general purpose breadth-first routing.

TML specifies the token and the handlers. A token can have multiple instantiations termed as subtokens with separate private memory allocation, but the same handler. A token execution may not be atomic, if it has a subroutine call inside it. Such a subroutine is called a subcall. A subcall is executed in implicit split-phase calls using a continuation passing style (CPS) transformation. The subcalls enable construction of complex token methods at the cost of efficiency provided by atomic execution.

TML/DTM has been successfully implemented and deployed. The integrations of standard distributed services (gradients, leader election, etc) into the TML/DTM architecture were smooth and simple, validating the claims of the authors.

The running time of atomic executions of token handlers is hard to estimate beforehand because of their dependence on TinyOS event handler. Improved execution model is needed to provide more precise timing. Explicit coding is required to ensure the subcalls' computations finish quickly in a short bound time. More efficient memory management also needs attention to reduce the memory overhead due to the tokens and handlers.

2.2.4 KAIROS [20]

Wireless Sensor Network programming abstraction (allowing the programmers to present the global behavior in a centralized fashion) makes the language more flexible (suitable for writing different types of application) and simplifies application development (hiding low-level detail from the programmers).

Kairos presents a macroprogramming paradigm for sensor network programming. Kairos' programming model uses a centralized approach to present the global view of a distributed application running on a sensor network. The abstraction hides from the programmer inter-node communication and program coordination across the nodes. As the abstraction is not node-specific, it focuses more on expressivity rather than performance tuning. Algorithms written in Kairos are compact and flexible resulting in the development of robust applications.

In Kairos, program code is generated by a language pre-processor, which is then compiled by the architecture specific compiler. This makes Kairos language-independent. The compiler produces a node-specific version of the distributed program. The Kairos runtime library at each node provides local variable abstraction to the remote state variables.

Kairos has adopted declarative programming model where the programmers are provided with three abstractions:

- Node Abstraction: Any node across the network can be accessed through a local integer identifier. This abstraction hides the complexities of network topology discovery from the programmers.
- Remote data access: The detail of underlying message communication is hidden from the programmers by making the remote and local variables appear as local in the program (shared memory abstraction).

- One-hop neighbors: The process of neighborhood discovery is hidden from the programmers, who can access all one-hop neighbors through a simple function call.

Kairos guarantees eventual node state consistency. This allows nodes to become consistent over a period of time. This may result in a node having a stale value, but eventually, after a bound time interval, its value will be updated. Loose synchrony blocks a reading if the referenced remote variable is not valid.

Kairos has been successfully implemented and deployed. Evaluations have been done based on experimentations with three different distributed applications. The results consistently validate the claims of Kairos. The middleware provided by Kairos is not yet equipped to control the underlying run-time resources, or to optimize any application specific communication pattern.

2.2.5 TASK [24]

TASK is a system designed for use by naïve end-users to deploy and manage sensor network applications with nodes running TinyDB. The TASK sensor kit is made for 'turnkey' sensornet applications for the Berkeley motes (mica2 or mica2dot). It is designed specifically for low data rate environmental monitoring application. This kit is meant for users who are not sophisticated computer users. Thus, the main focus of this kit is to make the deploying of the sensor network simple, as well as, it should be easy to configure, and easy to maintain.

The TASK kit is a three-tier architecture.

- The top layer consists of the TASK user tools, through which the users can interact with the sensor nodes in the network.
- The middle tier is what is called a Sensor Network Appliance (SNA).
- The bottom layer is the collection of sensor nodes running TinyDB.

TASK provides two client tools, one being a simple web based tool and the other, an advanced Visual Basic based tool. In addition to the services provided by the simple tool (client interaction activities as described above), the latter provides the user with a visualization of the deployed network with continuous health monitoring. Also, there are field tools, where a user can issue commands through a PDA in the mote neighborhood to check the health of the mote. TASK can also integrate with external tools like Matlab, Excel, etc.

SNA acts as a portal between the clients and the nodes. SNA is a resource enriched base station (Stargate platform from Intel Research and Crossbow), consisting of a local DBMS and a TASK server. The DBMS collects data from the sensor nodes and store them locally. The TASK server provides a web interface to the clients for submitting commands and sensor data queries through the TinyDB interface, monitoring network health, and browsing the local database through a standard ODBC interface.

TinyDB in the motes presents the network as a virtual database containing sensor data for all the sensor motes. The TASK server queries the database using TinySQL, and the client commands are processed by a command interface supported by TinyDB. One major contribution of this research is to add some improved features to TinyDB, which was then not ready to support real time network requirements. Efficient power management solutions to TinyDB have resulted in prolonging the lifetime of the network, which is often desired, for environment monitoring applications. Time synchronization features have been added to synchronize the node functionalities. The novel query sharing method guarantees that a node will not miss a query. Watchdog and data logging features along with query sharing increases network reliability by providing fault tolerance features.

TASK has been successfully implemented and deployed and the authors had performed several experiments to analyze its features. The results were satisfactory, reflecting the improvements to TinyDB. Also, the interface provided by TASK was easy to use by the naïve users. The drawbacks of TASK include its limitation to low-data environmental monitoring applications and laborious and time-consuming process of calibrating the sensors individually. Also, though, TinyDB is a very advanced distributed query processor, even if with efficient power management it remains power hungry, which is a major limitation to the network longevity.

2.2.6 TENET [28]

Tiered architecture increases system manageability (by implementing multi-node data fusion functionality and multi-node application logic in the master tier) and 'tasks' simplifies application development (by allowing code reuse). This paper introduces TENET, two-tiered sensor network deployment architecture. The lower tire constitutes of sensor nodes, and relatively powerful 32-bit platform nodes constitute the masters. The masters send commands to the nodes for doing in-node computation on local sensor data. The masters then collect the processed data from all the sensor nodes and do resource intensive computation on them, e.g. data aggregation, multi-node application execution. This way, the sensor nodes are not needed to do the resource demanding executions, and the average lifetime of the network is thus, increased. The commands send by the masters are encapsulated in 'tasks'. This supports code-reusability. Due to greater capacity of the masters, network coverage is considerably extended, supporting scalability. TENET also provides a reliable and efficient underlying networking for the communication between the nodes and the masters.

TENET is a two-tired software architecture comprising of the Sensor nodes and Masters.The sensor nodes supported by Tenet are MicaZ and Tmotes. The motes cannot initiate computation on their own, but wait for tasks from the masters for execution. The motes run the TinyOS operating systems to take advantage of its reliable drivers, including timers, sensors, etc. The tasks are usually limited to small, simple applications to reduce the computation overhead of the motes. Tenet scheduler dynamically schedules tasks in a mote and improves system efficiency by allowing on-demand memory allocation. The master nodes are fewer in number compared to the sensor nodes. Masters are powerful 32-bit nodes like Stargates or PCs. Masters can communicate among themselves to execute the multi-data and multi-node related programs.

Tasks are small programs written in a language created by the Tenet researchers for this architecture. A task comprises of a sequence of smaller tasklets. Each tasklet is a program providing service as part of a task. The tasklets are stored in a task library. The task library comes with the Tenet architecture package. The different tasklets are composed to construct program for data acquisition, processing, filtering, management tasks, etc. These programs can be reused for different applications. This helps Tenet to run different applications in the network concurrently. Application development requires combining the tasklets to construct different tasks. Available task library, modular programming style, and the simple and expressive task language makes application development simpler for the programmers.

Communication is message oriented comprising of two functions. (1) sending tasks to the nodes from the masters and (2) sending back task responses from the nodes to the masters.

Tenet uses a robust, scalable, tiered, data-driven routing mechanism. In this mechanism, a node sends task response to the nearest master, which in turn sends the data to the rightful receiver. Three types of response mechanisms supports reliable delivery of data for low data rate applications, events and high data rate applications. The tasks are broadcasted to the nodes using a reliable flooding protocol. The subsystem supports diverse applications.

Tenet has been successfully implemented and deployed. The novel features of Tenet have been evaluated through two application case studies. The evaluation proves the claims of Tenet but at the cost of communication overhead, delayed data processing, and possible network congestion. Time needed for the masters to receive the task response may delay data processing, reducing real-time efficiency. As in-node data aggregation is not taking place, a large amount of raw sensor data in the network can result in congestion. But the contribution of Tenet overshadows these drawbacks.

2.2.7 FACTS [29]

FACTS is middleware programming providing a rule-based programming language, with an underlying event-centric architecture makes WSN programming simpler, while attempting to minimize resource utilization. In this paper the authors have introduced a middleware abstraction layer, named FACTS, which combines the advantages of both event-based model and the rule-based model.

- Event-driven model is suitable for sensor network applications, as any change in the surrounding environment (events) requires prompt response.
- Rule-based models cannot provide such real-time response, but can provide a high-level abstraction to the programmers by hiding the underlying event coordination complexities and the communication detail. This abstraction makes it easier for programmers to develop complex applications for diverse distributed sensor network systems.

The main abstractions designed in FACTS are rules, facts and functions. Facts include sensor data and local and shared variables. The rules follow the guarded-command model. A guard is a condition based on the facts. If the condition evaluates to true, its corresponding action is executed. The action is responsible for changing the facts in a process. Also, the rules can call system functions, which are event-based. Each node has its local set of rules, facts and functions.

Rules are managed by an entity called the rule-engine. Rule engine is responsible for firing a rule or calling a function. A rule is said to be fired, if the guard condition is evaluated to be true. Rules can have priorities assigned to them, and are executed in an event-driven style to achieve fast response to events (real-time response) while economizing energy and memory usage. In a rule, an action corresponding to a guard is executed only if the guard condition is evaluated to be true as well as there is a change in the value (compared to the value in the previous round) of one of the facts involved. The latter condition makes the rule execution event-driven. The modified facts appear as events to the programmers.

Facts can be local or remote. An abstraction hides the sharing of facts (radio communication detail) from the programmer to whom both local and remote facts appear as local. Thus, the programmers are not required to deal with the complexities of sharing of information among the nodes, dynamic addition or updating of the facts, or the events triggered by modification of the facts.

Rules can call system functions through rule engine. The functions implement resource-aware low-level event-driven algorithms for real-time sensor network applications. The middleware offers a rule-based interface to the programmers with an underlying event-centric architecture accessible to the programmer.

The FACTS middleware has been successfully implemented. To illustrate the features of FACTS, the author has presented multiple examples covering major tasks typical to a sensor network system. The examples show how FACTS' implementation is resource-aware while presenting a simple high-level interface to the programmers. The programming structure is not very modular (component-based), which affects the runtime performance of the FACTS rule engines.

2.2.8 Rule-Based Programming Language [34]

Rule-based wireless sensor network programming language makes application programming simpler (by hiding the low-level detail) and eases program correctness proving, while being resource aware. In this paper the authors have proposed an intermediate programming language, which provides a high-level abstraction to the sensor network programmers. Unlike NesC, the language provides diverse high-level abstraction to the programmers, by hiding the low-level event detail. The compilation process provides reliability while economizing resource utilization. The code is compiled to an intermediate byte-code to be executed by any virtual machine, independent of the TinyOS/NesC platform. The paper also proposes an efficient communication protocol to exchange messages among sensor nodes. The applications supported by this language ranges from distributed data collection to reading sensor data for in-network computation and actuation.

An important contribution of this language is it being a state-based language, as opposed to the event-based ones. This paper explains how state-based model considerably improves reliability and power utilization of a process. In this state-based language, the guarded command execution style has been adopted. A set of rules and their corresponding actions are grouped together as a task. A rule is a condition based on the local and shared variables of a process, or the result of an event. When the condition is true, its corresponding action is executed. An action changes the state of the process by changing the variable values, but it does not directly trigger any event.

Each task is executed periodically till the program ends, and in each task the rules are evaluated in a sequential manner. The duty cycle of each task is parsed to ensure no overlapping. Thus, there is a single workflow in the process at any time. This ensures there is no threaded concurrency or dynamic scheduling, and hence, the workflow schedule is predictable. This simplifies the execution model compared to the event-based ones and at the same time allows the compiler to reduce the power usage like an event-driven model. Since the schedule is known during compile time, Weakest Pre-Condition program correctness proving methods, as well as self-stabilizing verification algorithms can be applied during compilation to check the program for any errors. This considerably increases the reliability of the program. Also, if task scheduling is violating any scheduling rule, like overlapping tasks, the compiler will inform the programmer about it.

The language also incorporates some event-driven execution styles by assigning priorities to the tasks giving the highest priority to the task dealing with event results requiring fastest response.

The communication model designed to support this programming is based on a shared memory model. The nodes are allowed to share data among its local tasks and with other nodes through shared memory termed as a ‘channel’. To support heterogeneous networks (different nodes having different sensors need to execute different rules), a network can have multiple channels, each channel having a different scope and type based on the type of a sensor node and the program running on it. A TDMA-like mechanism has been adopted where a process (node) needs to acquire a radio channel to send messages. This asynchronous communication economizes power utilization. Routing mechanisms may be deployed for multi-hop data transfer. The same message is sent multiple times to avoid communication loss.

The authors have illustrated their language by designing programs for an application for a heterogeneous sensor network, involving land management and herding of livestock. The language has not been implemented yet to analyze the reliability and energy efficiency of the network.

2.2.9 Declarative Resource Naming [35]

Macroprogramming (resource abstraction and dynamic binding) simplifies application development while economizing resource utilization. Declarative Resource Naming (DRN) is designed as a macroprogramming language with an aim to allow the programmers to write a wireless embedded system application in a high-level language while being resource aware.

In the DRN philosophy, a sensor network application is centered around operations on variables and resources (e.g. cameras, light sensors, temperature sensors, etc). Traditionally, accessing resources locally or remotely can be very tedious, as resources are not referred to by the node ids, but by their runtime properties. This involves implementation of complex algorithms for resource access, resource discovery, and across the network communication, etc. To overcome this difficulty, DRN proposes an abstraction of resources to hide these details from the programmer, making application development simpler. But at the same time, DRN supports imperative programming, letting programmers write complex algorithms for efficient software implementation.

DRN allows programmers to access a resource through a variable abstraction. The variable mapping lets the programmer access a particular resource by simply referring to its different run-time properties (camera turned on or off, temperature greater than a certain value) as Boolean expressions. Thus, there is no need for an application developer to provide the algorithms related to resource management. These algorithms are in DRN's supporting middleware. Other properties of DRN include tuning parameters such as time-out, energy budget (to improve energy usage), etc.

To hide the low-level communication detail from the programmer, the network is programmed as a single abstract machine (macroprogramming) with all the local and remote resources appearing as local to the program. The resources can be accessed individually or a group depending on their properties. When more than one resource satisfy a given set of properties they are accessed in parallel, thus reducing the total access time. Also group access allows in-network processing like data-aggregation, thus reducing energy consumption.

Mapping of each resource to a variable is called resource binding. Since sensor networks are reactive to environmental changes, the resource properties can change dynamically. Since a variable maps to a resource satisfying a fixed set of properties, the resource(s) for a particular variable changes over time resulting in dynamic binding.

DRN supports dynamic binding and hides the related details from the programmers. Sometimes static binding is also needed for cases where a particular resource needs to be accessed, which no longer satisfies some previously matched properties. DRN provides provision for such static bindings as well. If a resource is lost, or not accessible for some reason, an access timeout results raising an exception, which the programmer can access. DRN has been presented as a concept, which is yet to be implemented.

CHAPTER 3

DESAL (DYNAMIC EMBEDDED SENSING AND ACTUATION LANGUAGE)

Our research work includes design and development of DESAL (Dynamic Embedded Sensing and Actuation) [42], an integrated programming language and middleware platform for developing wireless sensor-actuator network applications. DESAL is a state-based rule-based programming language built on the TinyOS/NesC platform. An important feature of DESAL is that it is a state based language with guard-action commands. There are established formal correctness proving methods that can work on guard-action formats to mathematically check a program for errors. Also, there is no hidden control context like events or interrupts. DESAL also offers dynamic binding where processes communicate with each other through state sharing. This hides the detail of message communication from the user. Another novel feature we have introduced in DESAL is a variable of type 'token'. The concept of token is commonly used in mutual exclusion algorithms, where a process with a token is allowed to do a certain job, while others wait for that process to get done, i.e. wait for the token. Struct and functions are also important features newly added to DESAL.

DESAL is designed for applications, which do not have hard real time constraints, like natural habitat monitoring, etc. The reasons are (1) that time granularity is coarse (it can only schedule rules with seconds, not milliseconds), and (2) the timing of variable sharing is not precisely known, because messages can sometimes fail and need to be resent. Currently, there are two implementations of the DESAL platform differing in terms of the standard language and runtime. The first implementation is developed here, at University of Iowa. The second implementation is developed at Clemson University.

Our contribution to the DESAL project includes, (1) addition of modules to the Iowa DESAL middleware implementation, (2) writing DESAL compiler based on Clemson's grammar, (3) modifying Clemson's package to produce low-level NesC equivalent of DESAL program, (4) writing codes to convert a DESAL program to an equivalent Java program, (5) introduction of new features to DESAL (structs, functions and tokens).

DESAL is an attempt to combine many services already implemented by the previous research works. Services like time synchronization, neighborhood management, message communication, providing high level interface and base station independent interaction are all integrated in a single middleware.

Key features of DESAL include: (a) rule-based and state-based programming (b) synchronized action scheduling via timed execution (c) neighborhood management (d) communication via distributed state sharing (e) dynamic binding. A brief overview of the features is given below. Later we compare a DESAL program with an equivalent NesC program to show how writing code in DESAL is much simpler. We also briefly compare the different features of these two platforms. In chapter 3 we have discussed DESAL language design in detail. It includes the grammar and syntax of the language followed by the explanation of the runtime architecture. DESAL production has been done through collaboration between Clemson University and University of Iowa. The two universities developed the same DESAL but their approach was different. They have the same runtime architecture, but implementation is different. In the last section we discuss the runtime architecture and the different implementations. The Clemson runtime architecture is implemented based on Java, whereas, the codes written for Iowa DESAL are in Python.

3.1DESAL Features

The important features of DESAL are: (1) State-based programming model (2) Timing and time synchronization (3) Neighborhood Management (4) Communication via distributed state sharing and (5) Dynamic Binding.

In the state-based programming model, an action taken is not event-triggered, but depends on the truth-value of a Boolean condition comprising of the state variables. The guarded-command syntax has been adopted here. A guard is a condition comprising of declared state variables. When a guard is evaluated to true, a single or a group of statements corresponding to that guard is executed. The execution of these statements changes the values of the declared variables, but do not invoke any events. Thus, in this model, declared state variables decide the computation, and not the low-level events, or any other values stored in the memory. The guards are periodically evaluated in the body of the program, separated from the event-driven system functions, like sensor readings.

Advantages are (a) High-level abstraction: Since, program computation involves only the declared variables, applications are written without any knowledge of the low-level event scheduling (which is architecture specific). This presents a declarative style of programming providing a high-level abstraction, making program development much simpler compared to that in the event-driven models. (b) Predictable workflow: The periodic evaluation of guards (and not events) comprises of the execution workflow when a program is running in a mote. Since the guarded command execution involves only declared variable changes, the compiler can statically determine the workflow. This helps the compiler to improve the runtime of the program (best-effort scheduling) as well as check the correctness of the program (e.g. using weakest precondition).

Programmers are allowed to fine-tune their application with user-defined periods for guard execution. This can provide the programmer with more control to write application-specific efficient codes.

Time synchronization mechanism forms the backbone of the DESAL architecture. This mechanism provides a global synchronized clock. It allows the nodes to wake up at the same time, do synchronized computation and communication with sleeping in-between. Advantages are (a) Power awareness: Sleeps in-between executions result in low power consumption due to low duty cycles. (b) Coordinated Actuation: Synchronized clock enables the same guard across the nodes to get evaluated at the same time (on a best-effort basis). (c) Data freshness: Shared variable values could be time-stamped in a complete implementation (not done in present DESAL).

DESAL programs have built-in access to the network neighborhood of the hosting device. Neighborhood management services, including node discovery and health monitoring, are performed automatically. Health monitoring provides added robustness to the network.

In DESAL, communication is expressed by sharing of state variables. The nodes in the network share their states (declared state variable values) with each other through a soft-state abstraction. The soft-state makes the local as well as the remote variable values appear as local variables to a node. This is done by copying the remote variable values to their corresponding images (local variables) in the node. The soft-state store, termed as the best-effort cache, is updated periodically. Soft-state abstraction hides the low-level communication event management (writing the message construction and interpretation functions, and the message send/receive events) from the programmers. This makes application development simpler and less error-prone.

For a particular node, binding mechanism (part of the supporting DESAL runtime middleware) maps the remote variables to their corresponding local images (variables) in the node. As a wireless sensor network is usually very dynamic in nature, the mapping may change frequently over time. This change is periodically recorded by updating the soft-state cache, and accordingly the binding is updated. The dynamic binding feature takes care of the changes in the network topology due to the entry/exit of the sensor nodes, communication failure, etc. Before a binding is made or updated, a series of operations, like health monitoring, are executed to validate the binding. Advantages are (a) Dynamic network support: Programming supports frequent changes in the network topology. (b) Abstraction: Dynamic binding keeps the dynamic nature of the network transparent to the programmer. Hence, application development does not need writing of additional functions to take care of the network topology changes.

Previous research works (discussed in the literature review) have investigated most of the above features. DESAL attempts to combine all these features to present a powerful and at the same time, a simple high-level sensor network programming language. The important feature of DESAL is that, it hides the implementation of these features from the programmers, which are automatically integrated in a program during compilation. The static construction of the above mechanisms can enable efficient usage of data structures and expert programming to exploit the advantages of NesC and TinyOS. This can result in efficient memory utilization and economic power usage.

3.2 Comparing DESAL With NesC

NesC is a sensor network programming language. It is an extension in C developed to program on TinyOS, supporting component-based and event-driven programming. The basic features of NesC include the following [6]:

Different components are ‘wired’ together to form the whole program. The components are executed in tasks. Tasks execute sequentially (FIFO scheduling) in a non-blocking manner, and events can preempt tasks.

The component interface is provided to present the list of functionalities (events and commands) offered by that component. The programmers access the interfaces in their program through those functions.

A single interface presents the complex interaction of components through the commands (functions) and events. A command is posted as a task, and its completion is signaled as an event. Typically, commands call downwards, towards the hardware level, while the events call upwards to the interface level. Tasks are non-blocking, while events are triggered by hardware interrupts.

Static checking of TinyOS interfaces and variables (static wiring, static memory allocation, no function pointers) help prevent errors and economize resource utilization. Static compilation enables better code generation and analysis, including compile-time data race detection.

Below we present the Blink application in NesC followed by the same program in DESAL.

Blink is a basic application that toggles the LEDs on the mote on every clock interrupt. The clock interrupt is scheduled to occur every second. The initialization of the clock can be seen in the Blink initialization function, StdControl.start(). Figure 3 shows the low-level wiring of the components.

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;}
```

Figure 3 Component Wiring

It is obvious from Figure 3, 4 and 5 (given below) that code writing in DESAL is much simpler than that in NesC.

3.3 DESAL Language Design

Figure 6 shows a sample DESAL program. An abbreviated DESAL grammar is listed in Figure 7. Trivial productions are omitted.

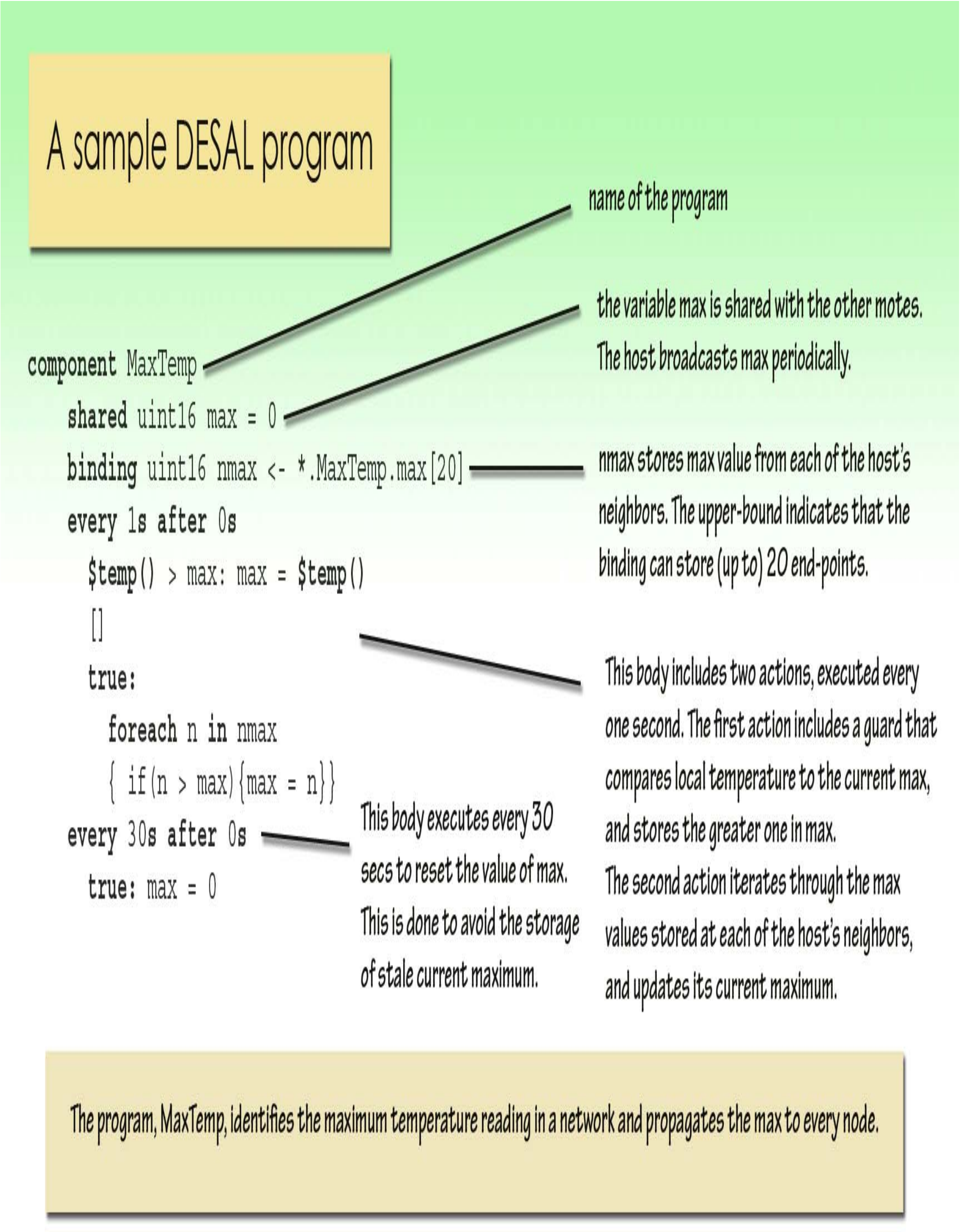
This section discusses Binding, Time Synchronization, Message Sharing and Neighborhood Management implementations for Clemson and Iowa. A picture view of the common runtime architecture is shown in Figure 8. The DESAL compiler translates a DESAL program to an equivalent NesC program. This enables NesC compiler’s code optimizations and the usage of the low-level TinyOS drivers.

```
/**
         * Implementation for Blink application.  Toggle the red LED when
         * a Timer fires.
         **/
        module BlinkM {
          provides {
            interface StdControl;
          }
          uses {
            interface Timer;
            interface Leds;
          }
        }
        implementation {
        command result_t StdControl.init() {
            call Leds.init();
            return SUCCESS;
          }
        command result_t StdControl.start() {
            // Start a repeating timer that fires every 1000ms
            return call Timer.start(TIMER_REPEAT, 1000);
          }
        command result_t StdControl.stop() {
            return call Timer.stop();}
        event result_t Timer.fired()
          {
            call Leds.redToggle();
            return SUCCESS;
          }}
```

Figure 4 Blink in NesC, taken from the application directory of TinyOS

```
component Blink
      every 3s after 0s
            true:
                $redOn()
      every 3s after 1s
            true:
                $redOff()
```

Figure 5 Blink in DESAL



A sample DESAL program

```
component MaxTemp
   shared uint16 max = 0
   binding uint16 nmax <- *.MaxTemp.max[20]
   every 1s after 0s
     $temp() > max: max = $temp()
     []
     true:
       foreach n in nmax
       { if(n > max){max = n}}
   every 30s after 0s
     true: max = 0
```

name of the program

the variable max is shared with the other motes. The host broadcasts max periodically.

nmax stores max value from each of the host's neighbors. The upper-bound indicates that the binding can store (up to) 20 end-points.

This body includes two actions, executed every one second. The first action includes a guard that compares local temperature to the current max, and stores the greater one in max. The second action iterates through the max values stored at each of the host's neighbors, and updates its current maximum.

This body executes every 30 secs to reset the value of max. This is done to avoid the storage of stale current maximum.

The program, MaxTemp, identifies the maximum temperature reading in a network and propagates the max to every node.

Figure 6 A sample DESAL program

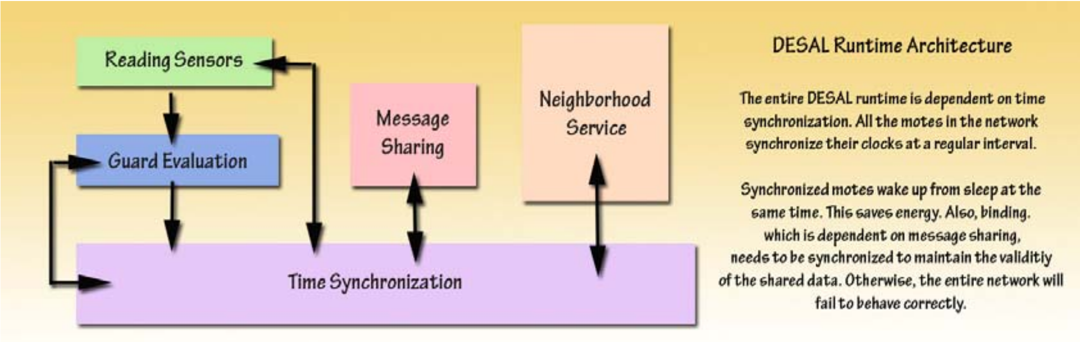
```
     1 program→ component <cid> <subcmpnts>
         component is the entire DESAL program. <cid> is the name of the
         component which is same as the file name.

       2 subcmpnt→ <vars> <bindings> <body>
         subcomponent consists of variable declarations, binding
         declarations and a body containing guarded actions.

       3 var→ [unshared|shared] <dec>
         variables which are private to the node are declared as
         unshared. The neighbors can read shared variables. Variables
         are shared through bindings.

       4 dec→ <type> <id>
         variables can be 8, 16 or 32 bytes. Small variable size helps
         in saving memory.

       5 binding→ binding (sbinding|mbinding)
         shared variables are bound to their corresponding image
         variables in the neighbors. Binding can be of two types:
         singleton binding and multiple binding.

       6 sbinding→ <dec> <- <nid>.<cid>.<vid>
         Singleton binding is used when only one particular neighbor
         reads a shared variable.
         <dec> is the name of the local copy of the shared variable
         <vid> in the component
         <cid>. <nid> is the id of the neighbor whose <vid> copy is
         read into <dec>.

       7 mbinding→ <dec> <- "*".<cid>.<vid>"["num"]"
         Multiple binding or multi-binding is used when more than one
neighbor reads the
         shared variable <vid> in component <cid>. This is indicated by
         "*". <num> puts an upper limit to the number of neighbors to
         avoid resource exhaustion.

       8 body→ every <num1><tunit> after <num2><tunit> <gactions>
         body consists of the guarded action commands <gactions>. Body
         is executed for <num1> time units after a delay of <num2> time
         units. The execution of the body is synchronized with body
         execution in other nodes with the help of underlying time
         synchronization.

       9 tunit→ ms|s|m|h
         time unit can be millisecond, second, minute, hour.

       10 gaction→ <guard> : <stmtlst>
          A guarded statement <stmtlst> is executed when the <guard> is
          enabled.
       11 guard→ <boolexpr>
```

Figure 7 DESAL grammar rules

```
   When a guard is enabled, <boolexpr> is true.

12 forstmt→ foreach <vid> in <bid> {<stmtlst>}
   For each statement is used to iterate through the copies of
   the shared variable <vid> in the multiple binding <bid>. The
   order of iteration is non-deterministic. Values for the failed
   neighbors are pruned automatically by the underlying
   neighborhood management.

13 bndfunc→ bound(<bid>)|src(<bid>)|age(<bid>)
   bound() returns a Boolean value indicating whether the binding
   end-point is healthy.
   Src() returns a node identifier of a binding end point.
   Age() is used to determine the freshness of a binding value.

14 sensfunc→
   $temp()|$humid()|$tsr()|$par()|$adc0()|$adc1()|$volt()
   The sensor functions are provided to retrieve sensor data.

15 actfunc→ $redOn()|$redOff()|$redToggle()| //..analogous
   for blue, yellow..
   Functions provided to control actuated components. The
   functions shown here correspond to the standard LEDs.
```

Figure 7 continued

3.4 DESAL Runtime Architecture



DESAL Runtime Architecture

The entire DESAL runtime is dependent on time synchronization. All the motes in the network synchronize their clocks at a regular interval.

Synchronized motes wake up from sleep at the same time. This saves energy. Also, binding. which is dependent on message sharing, needs to be synchronized to maintain the validitiy of the shared data. Otherwise, the entire network will fail to behave correctly.

Figure 8 DESAL Runtime Architecture common to both Clemson and Iowa implementation

DESAL also provides a middleware, which takes care of the Binding, Time Synchronization, Message Sharing, Neighborhood Management, Body Execution and Sensor Readings.

3.4.1 Clemson DESAL Runtime Architecture

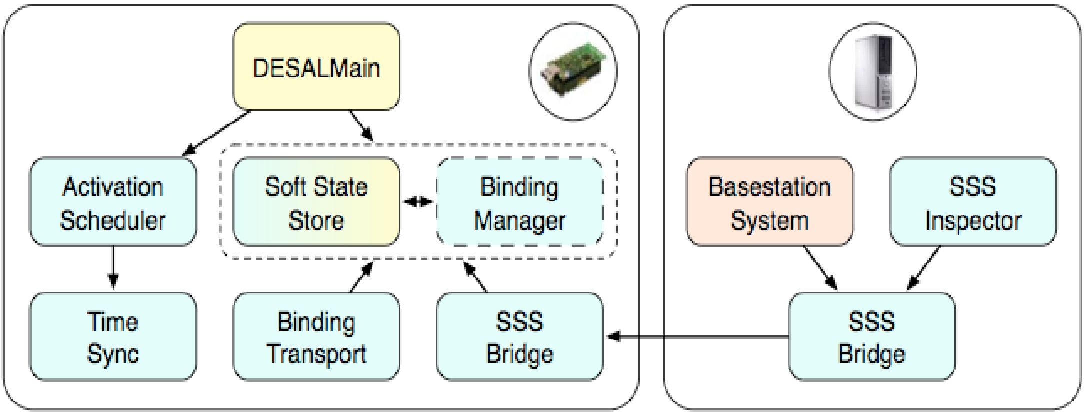


Figure 9 Clemson DESAL Runtime Architecture

Figure 9 shows the visualization of Clemson DESAL runtime architecture. Blue modules (Activation Scheduler, Time Sync, part of Soft State Store, Binding Transport, Binding Manager, SSS Bridge, SSS Inspector) are common to all applications, yellow modules (part of Soft State Store, DESAL main) are compiler- generated, and the edges between them represent dependencies. The Binding Manager is a conceptual module introduced for the sake of exposition. Its functionality is implemented by the Soft State Store. The orange module (Base Station system) represents a Java application running on a serial-attached base station.

3.4.1.1 Binding, Message Sharing And Neighborhood Service

The unshared DESAL variables are translated to corresponding NesC variables. The shared variables are not directly stored as NesC variables. Instead they are stored in the memory termed as the Soft State Store. The shared variables and bindings are translated to handles used to interact with the Store. The shared variables of a node are periodically broadcasted to the neighbors. The Soft State Store is updated regularly (When a variable is altered, the Soft State Transport is notified.) to reflect neighborhood changes, which includes the event of a neighbor leaving the network, a new neighbor joining the network, or a neighbor's shared variable value getting modified. The Binding Manager implements the DESAL binding functions (i.e., bound(), src(), age()) using the end-point data maintained by the Store. The module is also responsible for pruning failed end-points from multi-bindings. To amortize the pruning expense, the end-point check is performed as part of the iteration process: Each time the current slot in a multi-binding is set to the next slot in an iteration, the age of the entry is compared to its allowable lifetime. The latter value specifies the maximum time an end-point should be considered active without receiving an update. (The value is set as a compile-time option.) When the age of a slot exceeds its lifetime, it is pruned. The Soft State Store Bridge provides a communication interface between a DESAL device and a serial-attached base station. The Bridge consists of a NesC module and a corresponding Java class library. The Java library provides base station services to inspect and modify the elements managed by the Store of the attached device. It can, for instance, be used to establish a virtual binding across the serial link. The Soft State Store Inspector is a simple graphical interface for debugging, constructed using the Bridge library. It enables developers to easily monitor the contents of the Store and to inject changes in variables and bindings.

3.4.1.2 Time Synchronization In The Implementation

The time synchronization forms the backbone of the DESAL architecture. The time periods needed for the execution of the body (guards and sensors) and the Soft State Store (and Neighborhood service) are provided by the host’s clock. To synchronize activations across the network the time synchronization protocol uses a beaconing approach that converges to the lowest clock value in the network. Thus, all the different components of the runtime architecture are dependent on the Time Synchronization module. Synchronization is important for two main reasons: (a) this maintains the validity of the shared variables. e.g. for calculating the average temperature of the network, the temperatures from all the nodes in the network should be collected approximately at the same time to produce the correct result. (b) Synchronization helps saving network energy. Nodes periodically go to sleep to conserve energy. For correct functioning of the network, the interacting nodes should wake up at the same time.

3.4.1.3 Body Execution And Sensor Reading

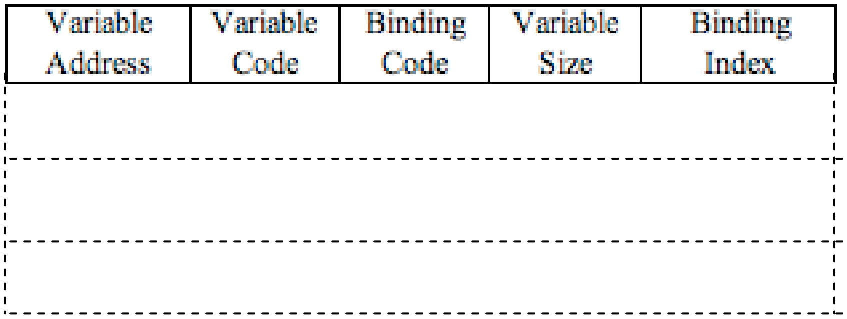
Each subcomponent body is translated into a function with the guards and actions as if-then blocks. A body is executed periodically according to the specified periodicity and delay provided in the DESAL function. The sensor values are also read periodically. Each body is executed as a separate thread using the Tiny Thread Library for TinyOS [4]. Using the Tiny Thread approach, each body/function is translated to a single blocking call on a wrapped driver. Threads are activated based on the periodicity and delay parameters specified by their corresponding subcomponent body. These activations are handled by the Activation Scheduler. The module provides an interface for requesting activation events at a specified periodicity, after some initial delay. At boot time, an activation schedule is requested for each thread. Internally, the module uses one-shot timers tuned to the system clock. The timer delay for a given thread is set to (time+delay)%period, where time denotes the current value of the clock, delay represents the requested delay, and period represents the requested period. When the timer fires, the activation event is signaled on the body thread, and the one-shot timer is again set according to the same formula.

3.4.2 Iowa DESAL Runtime Architecture

3.4.2.1 Binding

Sensor motes running DESAL program periodically share messages with its neighbors. The motes exchange messages through shared variables. There are two types of shared variables: internal and external. The external variables are writable, and only the host mote can write to it. The host periodically broadcasts the values of these external variables. The motes in need of those values receive and copy those variable values to their corresponding images. The images are actually shared variables to which those received values get copied. These variables are termed as internal shared variables. They can be thought of as internal images of some external variables whose value is received via the radio. These internal variables are readable only, as they are just images of some external variables. The bindings of the external variables to the corresponding internal ones are declared in the Binding section of the DESAL program. Below we discuss parts of the Binding Table.

For each shared variable, the compiler creates a binding tuple as given in Figure 10. All the tuple constitute a Binding table. The field Variable Address has the address of the variable. Variable Code is a unique integer number given to each shared variable. In the current version of the DESAL program, this code is actually the (incremental) serial number for a variable. Binding Code is assigned to an internal shared variable during runtime when it gets bound to the corresponding external variable (this happens when the variable value is received for the first time). Next time onwards, the received value gets directly copied to this bound internal variable. As the name indicates, the Variable Size field contains the size of the shared variable. Binding index contains two types of information: the first three bits of the byte indicates whether the variable is Bound, Writable, or a System Variable.



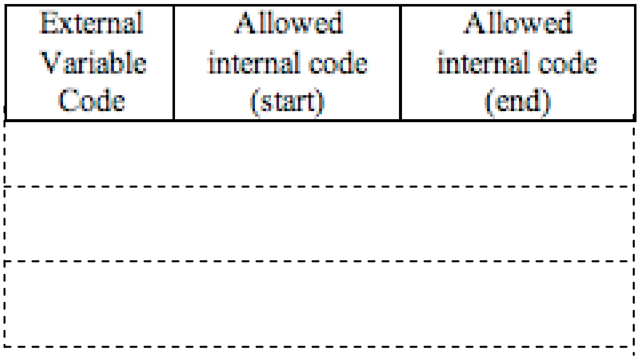
Variable Address	Variable Code	Binding Code	Variable Size	Binding Index

Figure 10 A Binding Tuple associates an internal variable to an external variable. All the tuple constitute of a Binding Table. The first variable address contains the address of the host/external variable. The first Binding Index is the index of the external variable, which is zero. The next (n-1) tuple corresponds to the internal variables from the (n-1) neighbors.

If the variable is bound, the next 5 bits give the id of the neighbor to which it is bound. This is done during runtime when the binding is done for a particular variable. This table also gets modified during runtime to reflect the changes in the bindings when the neighbors leave or enter the system. How the Binding table is constructed is explained below with an example.

Suppose in a sensor network of n motes, all the motes are running the same DESAL program. Each mote has a shared variable called x, which it shares with all its (n-1) neighbors. Then, according to the program, x will be an external shared variable as its value is getting broadcasted to the other motes. Also, there will be another (n-1) internal shared variables, which will be the corresponding images of the external variable x coming from each of the neighbors. Thus, the Binding table will have n entries, the first tuple corresponding to the external variable, and each of the next (n-1) entries for the internal variables. Initially, before the motes start interacting, the Binding code correspond to each variable will be zero, which means it is not bound yet. The compiler also builds a table called the Allowed Binding Map shown in Figure 11. Each (n-1) tuple in the table corresponds to an external variable.

The first field contains the variable code. The next two fields store the variable codes of the shared variables specifying the range of the internal variables to which that external variable can be bound. How this mapping works can be explained with the previous example. The program contains a single external variable called x. The mote will be receiving (n-1) copies of this variable x, each coming from a neighbor. So we say the (n-1) values (images) received over the radio can get mapped to the corresponding (n-1) images of the external variable x.



External Variable Code	Allowed internal code (start)	Allowed internal code (end)

Figure 11 Allowed Binding Map Table contains the external or the host variable code, and the allowed internal variable codes contain the range of the codes allowed for the variables from the neighbors.

These images get stored in the corresponding internal variables. The first field in the Allowed Binding Map table contains the name of the external variable, and the next two fields contain the variable codes of the first and the last internal variable specifying the range of the internal variables to which the value of the received x can be copied.

Now we talk about the Binding Action. We need to specify which internal variable will contain the image from a particular neighbor. Since all the internal variables corresponding to x are identical, if we don’t fix a variable for a particular neighbor, each time a value is received from a neighbor, it can get copied to a different internal variable. So, for a particular neighbor, we need to fix one of the (n-1) different internal variables. This process is what is known as binding. When a value from a neighbor is received for the first time, it means the value is not yet bound to any of the allowed internal variables. At the same time, the Binding table will show binding information for internal variables which are bound to some different neighbors, as well as some unbound variables, one of which is allowed to get bound to the said neighbor. To determine the code of the internal variable to which it can bind to, the following steps will take place: (1) the allowed binding map table will be checked to find the allowable range of internal variables. (2) Among the internal variables an unbound internal variable will be chosen. (3) Then the binding code and the binding index will be updated. Thus, next time a value of that external variable is received from that same neighbor, the binding information is matched from the Binding table, and it gets copied to the same internal variable.

The Figure 12 is showing a sensor network with four motes. Let MT denotes the allowed binding map table, and BT the binding table for the mote with id=1. Mote 1 has three neighbors with Ids – 2, 3 and 4. Each has a shared variable x which is getting shared with the neighbors. In BT the first filled column represents the internal variable codes, and the last column represents the neighbor Id to which that variable is bound.

The binding protocol does not allow overlapping of an existing binding. This means, if an internal variable is already bound to an external variable from another mote, the binding remains till the neighbor leaves the network.

3.4.2.2 Message Sharing

The program periodically broadcasts the shared variables. The period has been fixed at EXCHANGE_PERIOD (= CYCLE_PERIOD/3). CYCLE_PERIOD is the guarded command period. The guards get executed every CYCLE_PERIOD. At the wake of the timer, a TOS_Msg is created. The payload part of that message is shown in Figure 13.

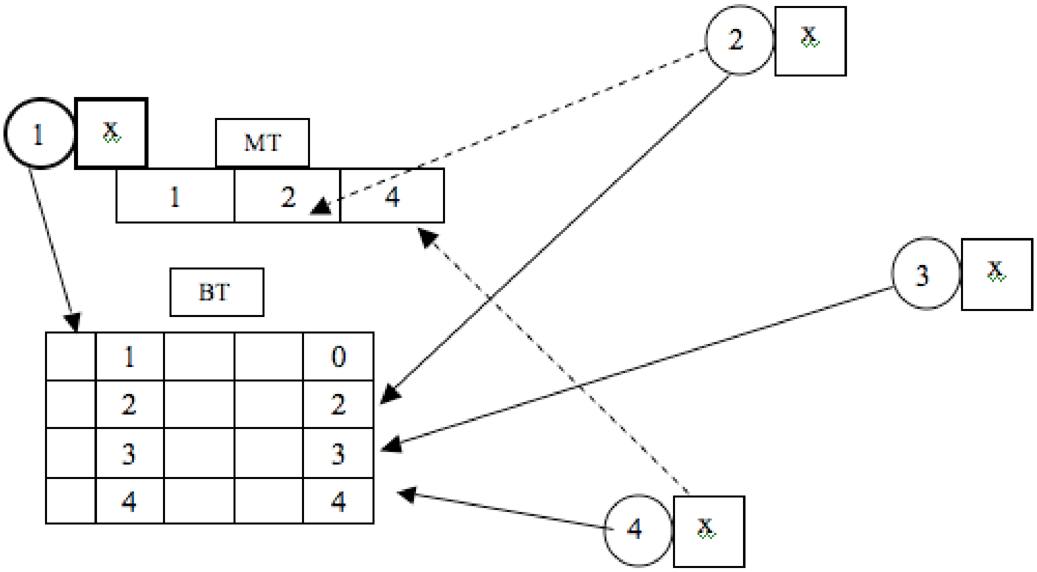


Figure 12 Binding Action: Here 1 is the host node with neighbors 2, 3, and 4. MT is allowed binding map table. BT is the binding table. In MT allowed binding range of the external variables is 2 to 4 as the internal variable codes range from 2 to 4. In BT the first index in the last column is 0, which is the binding index for the external variable. The rest of the indices correspond to the id of the neighbors.

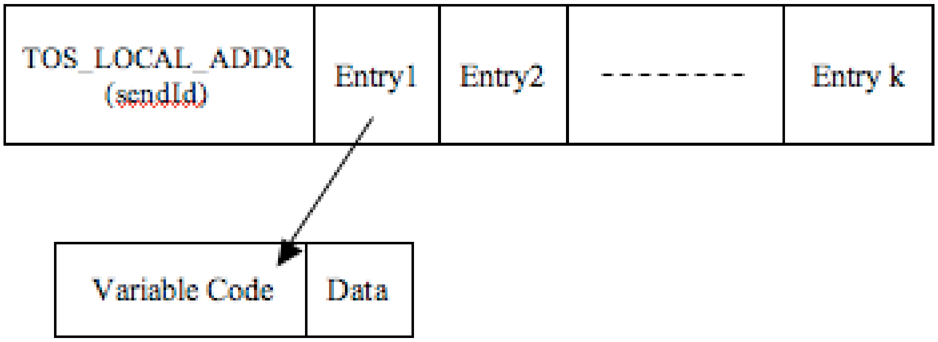


Figure 13 Message Payload is the tuple, which is broadcasted over the radio. The first field contains the id of the sender. The rest of the entries contain the external variable values to be broadcasted. With each variable value (data) the variable code is also attached.

The payload has two parts: the SendId of the sender, and the set of entries to be sent. Each entry consists of the Variable code, and the corresponding data. This code is the external variable code associated with that variable. If the variables are system variables they are not shared. In a round, it may happen that all the variable data do not get fitted in a single TOS_Msg. In that case, more TOS_Msgs are sent in a round till all the variable data are sent. These messages are sent at an interval of 1/8 sec. The entire set of TOS_Msgs is sent at an interval set at EXCHANGE_PERIOD. If sending a message fails, sending is retried after every ¼ sec. After a TOS_Msg is received, all the entries are processed one by one, and the data corresponding to each external variable code gets copied to the corresponding internal variable.

The variables read by a neighbor through message sharing is accepted by binding the variables to the corresponding images, i.e. the internal variables. After receiving the variables the node cannot modify them. This way, the neighbors communicate with each other through state sharing. The mechanism of the message sharing is hidden from the programmer. While writing the code the programmer works with only the internal and the external variables.

We have worked with a network, which is dynamic in nature. Hence, neighbors can enter or leave the network anytime. This is reflected in the binding table. When a neighbor joins the network, a new tuple corresponding to the new neighbor is created. Also, the range in the allowed binding map table is modified. Similarly, when a neighbor leaves the network, the tables again get modified. Thus, we have dynamic binding. How the network changes are recorded is described in the neighbor service, which we discuss next.

3.4.2.3 Neighborhood Service

When a neighbor enters the system, an event is triggered in each of the motes already in the network. The event checks the neighbor Ids array to find if the neighbor already has an entry or not. If there is none, the neighbor Ids array is updated with the new entry. Neighbors joining the network cannot have Id=0, which is reserved for the Base. When a neighbor leaves the system, an event is triggered in each of the motes already in the network. The event look up the neighbor in the Binding table and sets the Binding Index to the default value zero. Then it clears the neighbor entry from the neighborIds array. Multiple messages are exchanged between two motes before they decide to become neighbors. The messages are exchanged to synchronize with the neighbor, to determine if the neighbor is stable and to determine if a neighbor has left the network. Each mote maintains a history table to store the messages from each neighbor (1 byte in size). The values of the history table are checked to verify if a neighbor is healthy or not. A neighbor is declared unhealthy if it has become unreachable from the host, or if it has stopped sending any messages. The health of a neighbor is checked periodically based on the host's clock. The Neighborhood functions are dependent on the Time

Synchronization module. The neighbor data structure (history table) is updated at a specific time interval, which reflects the changes in the network neighborhood.

3.4.2.4 Body Execution And Sensor Reading

In DESAL, guards are evaluated in the Body of a DESAL program. The Body is included in a task, called ruleExecute, in the corresponding NesC code, and the guards are evaluated in a switch statement. A guard- command pair is converted to an if-then block in NesC. In the NesC code, each guard has an index. Each index forms a case in the switch. When the task ruleExecute is posted, the cases are checked in a sequential manner. A counter called gindex holds the index of the current guard-command block being evaluated. This counter is the switch condition. The counter is incremented after the evaluation of each such block. If the value of the counter is less than the total number of guards, the task ruleExecute is posted again till all if-then blocks are evaluated one after the other sequentially. The time period between consecutive executions of the group of guards is an amount of time mentioned as a Body parameter (CYCLE_PERIOD). After the evaluation of all the guards in the sequential manner, the local time of the mote is synchronized with that of all other motes.

If sensors are defined in the program, the task prepSensor() gets executed which invokes sensor reading. The sensors are read at an interval set at the CYCLE_PERIOD. In a round, the sensors are invoked once, one after the other, and the respective getData() methods are called to read the sensor values. There is a pause of 1/8 second between the invocations of two consecutive sensor readings. After one round, the ruleExecute() task is called to evaluate the guards. Thus, the sensor evaluation task, followed by the guard evaluation task is repeated at a specific interval. If there are no sensors declared in the program, only the guard evaluation task is executed at an interval set at the CYCLE_PERIOD.

3.4.2.5 Time Synchronization

Time Synchronization module in the language is called after each round of sensor reading and guard evaluation. In one round the local clocks in the sensor nodes are expected to drift within the range of the CYCLE_PERIOD. So, during synchronization the local clocks of the motes are advanced to the start of the next CYCLE_PERIOD. This way, when the nodes wake up with the alarm clock, they are all synchronized to start the next round together at the same time.

Time Synchronization in the implementation forms the backbone of the runtime architecture. Synchronization is important for two main reasons: (a) this maintains the validity of the shared variables. e.g. for calculating the average temperature of the network, the temperatures from all the nodes in the network should be collected approximately at the same time to produce the correct result. (b) Synchronization helps saving network energy. Nodes periodically go to sleep to conserve energy. For correct functioning of the network, the interacting nodes should wake up at the same time.

```
event(timeout):
read global clock & prepare beacon message m ;
local-broadcast(m) to neighborhood ;
        schedule next timeout in φ seconds

event(receive beacon m):
c = read global clock ;
if m.timestamp > c then
        adjust global clock by + (m.timestamp − c) ;
```

Figure 14 Convergence to Max time synchronization protocol

For time synchronization, convergence to max protocol is used here. The idea is taken from the paper by Herman and Zhang [52]. The technique is presented here with read and adjust interfaces of the global clock and assuming time stamping of the beacon messages. Each node executes the programs in Figure 14 concurrently with a sensor application. The system's timeout mechanism is such that the timeout event eventually occurs every φ seconds, even for arbitrary initial state.

CHAPTER 4
TOKEN TYPE VARIABLE IN DESAL

A novel feature in DESAL is the introduction of token type variable. The concept of token is generally used in distributed mutual exclusion algorithms, where a process with a token can enter the critical section while others wait for their turn. A token travels around the network. Each process, when receives a token, changes its state, and then passes on the token to another process. This action goes on around the network. A classic example of token-based application is Leader Election in a distributed network. A token can act as a network health monitor [53]. It can go around the network monitoring the health of each process noticing if the process is working properly or not. If the process is malfunctioning, the token will move on to another process notifying it about the failure. In our work we have discussed a case study with tokens in a ring topology. Later we have also shown that this implementation can be applied to some other topologies as well. One important condition for a network with tokens is to self-stabilize, e.g. in the case of mutual exclusion algorithm, we need only one token to be present in the entire network. In an illegitimate state there can be more than one token present. The objective is to reduce the number of tokens to exactly one, and then maintain this legitimate state. If a fault occurs, the system should automatically revert back to a legitimate state. This can be achieved if the system is self-stabilizing. We have shown how a token ring, where a certain distance separates tokens, can self stabilize. In DESAL, the implementation of tokens is hidden from the programmer. The programmer will declare a token variable and use the token functions num(x) and pass(x) explained later. In our paper, Separation of Circulating Tokens [43], we have discussed tokens in detail. This is given in the next few sections in this chapter. The Separation of Circulating Tokens paper [43] gives us a motivating example of using tokens. In a wireless sensor network if selectively some sensors are active at a certain time and sensors in other nodes are idle, that saves energy.

This paper shows how to achieve this with tokens. We have illustrated the working of the token with another case study with RFID tags. In this example, there are three tags, and only one of them can be active at a time. A token variable can be used to achieve this. The next section starts with the idea of rotating sentry. Rotating sentries are token abstractions moving around in a network to balance the power consumption. The nodes with the tokens will be awake, while the others will be in sleep mode, or will have their sensors in the idle condition. After this we consider a network with a ring topology and a rotating sentry protocol. This gives us a ring where energy consumption is balanced to increase the lifetime of the sensors. This is followed by the separation of circulating tokens paper [43], which talks about self-stabilization of a ring with tokens.

4.1 Adding Lifetime To Sensors

The organization of this section is as follows: 4.1.1 discusses the rotating sentry problem. This is followed by the literature survey. In the survey we discuss two papers, VigilNet [49] and R-Sentry [50]. In 4.1.2 we discuss how to increase the sensor lifetime in a ring topology. 4.1.3 discusses our paper Separation of Circulating Tokens [43]. 4.1.4 shows DESAL program to increase lifetime of the sensors.

4.1.1 Rotating Sentry Problem

Design of a wireless sensor network heavily depends on the capacity of each sensor node, including memory, processor and power constraints. In this section we will discuss different research problems that have addressed the energy limitation or lifetime of a sensor network, most of them providing guaranteed coverage and connectivity and some providing robustness. An interesting question is how to make a wireless sensor network efficient in all aspects. Research have shown that it is not possible to take care of all the aspects to have an efficient wireless sensor network, which takes care of all the limitations. e.g. if high data delivery ratio is wanted, the network should be strongly connected to make it robust. But this means that there should be redundant sensors working all the time. Again, if there are more sensors transmitting at the same time, we need to avoid data collision. If most of the sensors are always on, the network life will be limited. Hence, a network protocol is devised based on the application of the network.

So, depending on different applications, different solutions have varied design assumptions and objectives. Wang and Xiao [44] have listed down different design objectives and assumptions. The major design assumptions include network structure, sensor deployment strategy, detection model, sensing area, transmission range, time synchronization and failure model. The main design objectives include maximizing network lifetime, balanced energy consumption, sensing coverage, network connectivity, data delivery ratio, quality of surveillance, scalability and robustness. But as we have discussed before, it is not possible to achieve all the objectives independently without compromising one for another.

4.1.1.1 Literature Survey

A wireless sensor network comprises of multiple sensor motes, which can be deployed virtually anywhere on earth including physically inaccessible places. The sensor nodes form an ad-hoc network, which can be monitored remotely. The energy supplied to the nodes is usually through batteries attached to them. But, since they are remotely deployed, it is not possible to recharge or replace their batteries. But, the nature of their applications demands from the sensor nodes to remain alive for a considerable time period. Unfortunately, this needs to be achieved with the severely limited energy supply. Researchers have and are still coming up with various solutions to this problem. Kumar et al [45] talk about using probability for each node to randomly decide whether to sleep or wake up for a certain time period, maintaining a specified coverage. This solution works with minimum message exchange, but is not robust against failures. Tian and Georganas [46] talk about a sensor node which has all the information of its neighbor. If its coverage area, called the sponsored sector, is completely covered by its neighbors, a node goes to sleep. But, this mechanism underestimates the number of sensors that can be turned off. In its sponsored sector a node only considers the neighbors in its coverage area, and ignores the other nodes further away, but overlapping its coverage area. Ye et. al. [47] talk about PEAS (Probing Environment and Adaptive Sensing). This works well in a very unreliable network. This protocol maintains the active working nodes at a distance c. If a node finds out there are no neighbors at a distance c on all sides are sleeping, then it will go to sleep for a certain period of time. This is repeated at each round. The drawback of this mechanism is that a working node may never go to sleep resulting in unbalanced energy consumption. This protocol emphasizes more on network connectivity. The goal of ASCENT (Adaptive Self-Configuring sEnsor Networks Topologies) [48] is to maintain a certain data delivery ratio. This means it focuses on connectivity and data loss rate. Based on these measures a node locally decides which neighboring node to activate, and when it can go to sleep. Like PEAS this protocol also does not guarantee uniform energy consumption, as working nodes may never go to sleep.

Now we will talk about how to deploy the rotating sentry problem in a sensor network to achieve energy efficiency or increased network lifetime. Rotating sentry refers to a network design in which some nodes, or sentries, are selected to be on active duty while others are either idle (only listening over the radio) or in deep sleep (none of the sensing, processing or radio is working) for a specified time period. After that time period is over, some other non-sentry node takes up the role of the sentry. Hence, the sentry is rotating. This design claims to be energy efficient, and ensures balanced energy consumption. We discuss a couple of papers on rotating sentries with different design assumptions and objectives. The main objectives include network connectivity, coverage and robustness. These two papers are carefully chosen to demonstrate how different protocols address different aspects of the network, and cannot address all. In sensor networks, due to the nature of the hardware, there exists a fundamental tradeoff between network lifetime and network service quality [50].

4.1.1.2 VigilNet

He et. al. [49] designed a hierarchical energy efficient surveillance system called VigilNet. The rotating sentry mechanism focuses on the energy consumption, which takes care of the surveillance quality and stealthiness. At the beginning of each round the nodes are synchronized and a diffusion tree is created for routing. Then there is neighbor discovery. During that phase a neighbor sends to the host its ID, whether it is a sentry or not, number of sentries covering it, and its location. The next phase is sentry selection. A sensor node locally decides to become a sentry if it is an internal node of a diffusion tree or if it finds out that none of its neighbors is a sentry or is covered by a sentry. A sentry is active to monitor events while the non-sentry motes are in a low power state till an event occurs. When an event occurs the sentry wakes up the other nodes in the region, and they start collaborative target detection. To avoid contention of multiple nodes wanting to become a sentry, each node uses a random backoff delay to transmit its SENTRY DECLARE message. During that time if it gets such a message from a neighbor, it cancels its plan to become a sentry. The backoff delay of a node is set inversely proportional to its residual energy. This balances the energy dissipation of the network. The backoff delay is also inversely proportional to the number of neighbors that are not covered by the sentry. The sentry selection also ensures that there is at least one sentry in each sensing range. Here we can see how this protocol focuses more on sensing coverage than communication coverage. Each node makes the sentry decision locally. Hence, globally it can result in a non-optimal number of sentries. But, this local method reduces extra message overhead (except neighbor discovery), which improves its stealthiness. In this paper the authors propose two different schemes, proactive control and reactive control, to control the sleep-wakeup cycle. In the proactive control the sentry node sends out sleep beacons periodically. A non-sentry node goes to sleep when it receives the beacon and stays asleep for a certain period of time. It wakes up after the sleep timer expires and repeats the same process. In the reactive control, the sentries do not send any sleep beacons to the non-sentry nodes. Instead they go to sleep and wake up according to a timer. The nodes also wake up when they receive a wake up beacon from a sentry. The reactive mode is stealthier as it avoids the sleep beacon overhead. But, the drawback of this mode is, due to a long period of no communication with the sentry, the non-sentry clocks may drift in course of time. This way the neighboring nodes may not sleep-wake in lock-step fashion. This will force a sentry to repeatedly send awake beacons to wake up a neighborhood. In their future work authors plan to design a more aggressive power management strategy with passive wake-up capabilities.

4.1.1.3 R-Sentry

Authors Yu and Zhang [50] claim to develop a sentry system, which can provide a continuous coverage through bound recoveries from frequent failures, while prolonging the lifetime of the sensor network. While on duty, the network has active nodes and redundant nodes, which are sleeping with their radios off. The redundant nodes wake up time to time to check the status of the active nodes. These redundant nodes are called sentries as they monitor the health of the active nodes. Now, to conserve energy, the redundant nodes should take turn among themselves to do the monitoring. Also, their taking turn should be synchronized since we do not want to have a long time gap between two wake ups which may mean no monitoring of the active node for that time period. To address this problem, the authors proposed a coordinated scheduling algorithm among all the sentries, and hence they are called Rotating Sentries or R-Sentries. In this paper the authors have assumed the grid-based coverage model. The grid points that fall within a node’s sensing area are considered covered by that sensor node. Nodes exchange the list of grid points it can cover with their neighbors, called GridList. With sufficient node density, there is a high probability that, out of the uniformly randomly deployed nodes, there exists a set of nodes that could collectively cover all the grip points in the network field. The other nodes are the redundant nodes called sentries. Every sensor node has a group of neighbors with overlapping communication or sensing capabilities. The sensing redundant set (SRS) consists of nodes that can cover the same grid point(s). When an active node fails it can be replaced by a node in the SRS. An active node groups all the redundant nodes in its SRS and calls it a gang. This group decides the replacement of the active node in case it fails. The explanations in this paper assume a redundant node serves only one active node, and all sentries won’t fail before the active nodes do. The initialization phase has a high message overhead. This involves gang discovery and schedule bootstrapping. After presence announcement exchange phase, every node stays active and starts a random backoff timer, collecting presence announcements from its SRS members. Next, it locally determines if all the points in its GridList are covered by the non-redundant SRS members. If yes, it considers itself redundant and broadcasts redundancy announcement message. If not, it becomes active. At the end of the bootstrapping phase, the active nodes calculate their gangs’ schedules and send the message to the redundant nodes. This is needed to achieve coordinated sentry rotation. Thus, the sentries now have their wake up times. A sentry node periodically wakes up as scheduled, to probe the active node. If the active node fails, all the sentries become active and resume the initialization process. Otherwise, they go back to sleep. The active nodes continue communication with the sentries to keep them synchronized. Thus, there is always some extra message exchange is going on to keep the sentries coordinated, but still the energy saved during the sentry sleeps is more significant. This protocol replaces an active node only when it fails and not because it is getting depleted of energy. Hence, the active nodes tend to die down quicker than the redundant nodes. This means, considering the active nodes also as part of the gang, the energy usage is not balanced. The larger the grid size, there are more sentries, and hence more energy is saved increasing the lifetime of the network.

4.1.1.4 Discussion

In the VigilNet [49], the main objective is to provide stealthiness and surveillance. So, the nodes are synchronized in each round with the base, and the connectivity of the network is maintained. Robustness is not a major factor here, assuming no frequent failures. That's why, in one round or cycle, if a sentry fails it is not detected till the next cycle starts. Also, any other changes in the network are detected only at the beginning of each cycle. In this model all the nodes can have the same role, and the job of the sentry rotates among the nodes to achieve uniform energy consumption. In R-Sentry [50], the main objective is to detect and repair failures. The protocol focuses on sensing coverage. The sentry nodes are always synchronized with the active nodes, to achieve coordinated rotation among sentries. This way energy consumption is balanced among the sentries. But the active nodes are replaced only when it fails. This way if an active node never fails it works till it is depleted of its energy. Energy depletion is recorded as a node failure, which wakes up the sentries to replace it. This way energy consumption is not uniformly distributed considering sentries and active nodes together. This is not a concern here as the main objective is to maintain the sensing coverage all the time in spite of failures. This way, we can see that each proposed solution only addresses a subset of network issues. In reality it is not possible to address all the issues at once. This is what we are going to see in our proposed rotating sentry problem in a ring topology.

4.1.2 Increasing Sensor Lifetime In A Ring Topology

Here we are considering a network with a ring topology and a rotating sentry protocol. The network has m tokens and n nodes. The objective is to balance the energy consumption of the network to increase its lifetime. To achieve that, we assume the nodes with the tokens will have their sensors active in a round (sentries), while all the other nodes will have their sensors in the idle condition waiting for the token. The notation and model is same as that in our paper ' Separation of Circulation Tokens' [43] discussed after this section. The tokens are always circulating with no deadlock. In each round a node receives a token. This way, the balanced energy consumption is ensured. Initially the tokens can be anywhere in the network, but the token distribution will eventually self-stabilize with the tokens certain distance apart from each other. We present the self-stabilization protocols with proofs. In this work we see that it is not possible to address all the aspects of the network as we have discussed before. So, our main focus will be to increase network lifetime by increasing the lifetime of the sensors.

In each round, working nodes never go to sleep. Others also don't sleep, but certain sensor operations are turned off. The token communication and synchronization among the nodes go on continuously. Selective sensing saves energy. The solution is robust to changes in the topology due to failures or entry or leaving of nodes. If the ring size changes, it again stabilizes within a finite period of time, provided the failures are time limited and the network doesn't become disconnected. To ensure that failure of one node does not disconnect the ring, we assume here the transmission range of the nodes are strong enough to get connected to next two nodes. As the tokens reach a new node in each round, energy consumption is balanced. The nodes are only involved in receiving and passing the tokens. Hence, there is no need for back off mechanism (in many energy saving protocols a node needs to broadcast its intention of going to sleep, so that no one else covering the same area goes to sleep). At the initiation there is the overhead of neighbor discovery message exchange. But, after that this protocol only needs to exchange time synchronization messages and tokens. There is no need to broadcast anything else. Since each node is only talking to its neighbors, it does not need to use strong transmission signals. This saves energy. Only when there is a failure, a node may need to use stronger signals to connect to new neighbors. This protocol has certain limitations. It does not handle failure of tokens. We have shown in our paper that the ring will self-stabilize. The idea is, each time a token passes through the node with a counter, it is separated from another token by at least a distance of C, where 0 to C is the range of the counter.

4.1.2.1 Motivation

One motivating application in our paper [43] is physical process control. As an example, one can imagine a closed network where some objects are conveyed from place to place, with some physical processing (loading, unloading, modifications to parts) done at each place. For the health of the machinery it may be useful to keep the objects at some distance apart, so that facilities at the different places have time to recharge resources between object visits. This can be formalized by Petri nets. The circuit of the moving objects is a ring for this example. With an unhealthy initial state more than one object can be together. The objective is to separate them by a certain distance. The token of a Petri net can represent physical objects. The formalism of Petri nets allows us to add additional places, tokens and transitions so that a self-stabilizing network can be constructed. Eventually, the objects of interest will be kept apart by some desired distance [43].

There is a simple case where separation of tokens can be enforced in an open network. Figure 15 shows how distance between tokens can be enforced almost trivially, by throttling the rate of tokens injected into the network.

4.1.3 Separation Of Circulating Tokens

Self-stabilizing distributed control is often modeled by token abstractions. The problem in this paper [43] is to ensure that a synchronous system with m circulating tokens has at least d distance between them. This paper [43] explores a mechanism based on timing information in a synchronous model. The sensor nodes with tokens are enabled to sense while sensing in other nodes is turned off. The separation between the tokens enables a node to 'rest' a while before activating its sensors again. This arrangement can save a substantial amount of energy in the network where node power supply is limited. In the paper [43], the problem is first considered in a ring where d (distance between tokens) is given whilst m (number of tokens) and the ring size n in unknown. A second problem is to maximize d when m is given and n is unknown. The challenge, as with all self-stabilizing algorithms, is that tokens can be initially be located arbitrarily and the variables encoding timers or other variables may have unpredictable initial values. The protocols are expressed with Petri net formalism in a ring topology.

Desired properties of a token circulation protocol are labeled as D1–D5 below.

D1. At any time, m tokens are present in the system.

D2. The minimum distance between any two tokens is at least d.

D3. A token moves in each step, from one process to a neighboring process.

D4. Every process has a token equally often, i.e., in an execution of k steps, for any process pi, there is a token at pi for k *m/n steps.

D5. Following a transient failure that corrupts state variables of any number of processes, the system automatically recovers to behavior satisfying D1–D4.

Failure may also change the ring size. This may happen if a sensor node fails to operate, i.e. in this problem stops sending messages, it means the node has left the ring network. As soon as a node leaves the ring, the ring changes its size. Our protocol is self stabilizing to that.

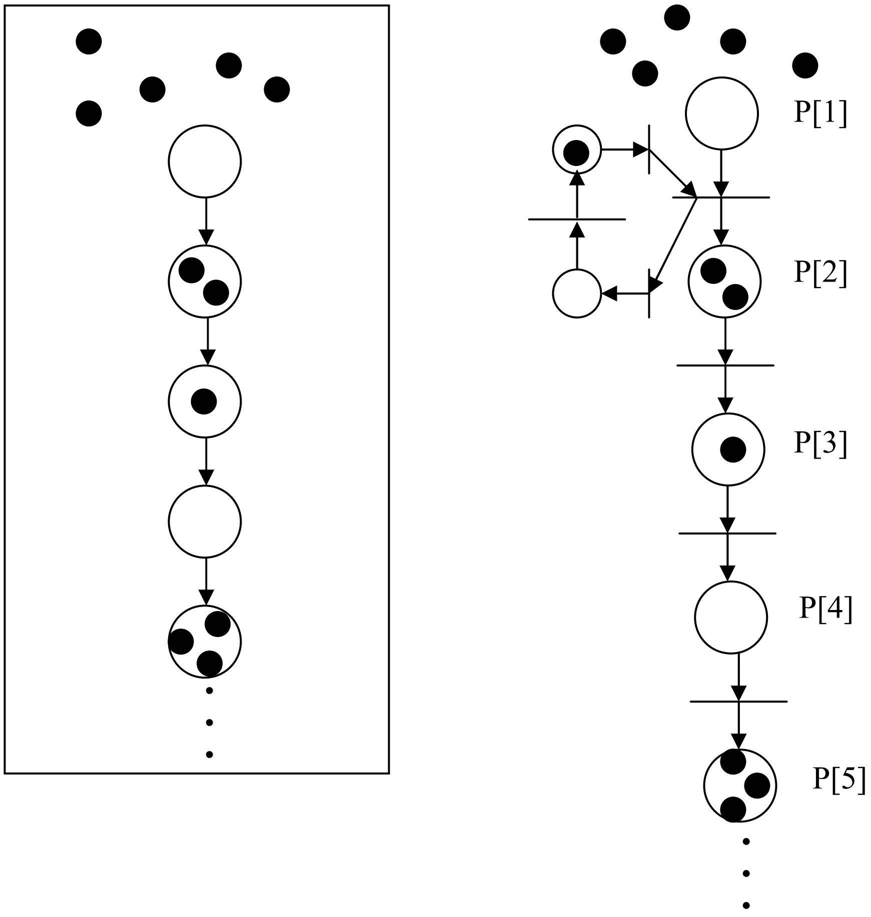


Figure 15 Illustrated on the left is an open system consisting of a chain of processes, p[1], p[2], …, with p[1] being the topmost process. Tokens arrive from outside the system to p1. Each process p[i] releases at most one token in each round to p[i+1]. The aim for this system is to ensure that, eventually, no two tokens are closer than some distance d in the sub chain from p[2] downwards (we cannot prevent the accumulation of tokens in p[1] in this open system). On the immediate right is a simple delay mechanism shown as a Petri net: the small sub ring and the joint transition between p[1] and p[2] ensures that the tokens do not arrive in each round to p[2]. By adjusting the size of the sub ring, the target distance d can be obtained.

4.1.3.1 Notations And Model

Consider a ring of n processes executing synchronously, in lock step. Each process perpetually executes steps of a program, which are called local steps.

In one global step, every process executes a local step. Programs are structured as infinite loops, where the body of a loop contains statements that correspond to local steps.

We assume that all processes execute the local steps in a coordinated manner. For processes running the same program, all of them execute the first statement step in unison. Similarly, if two processes run distinct programs, we suppose they begin the body of the loop together, which may entail padding the loop of one program to be the same number of steps as the other program. The execution of all steps in the loop, from first to last statement, is called a round. The notion of distance between locations in the ring can be measured in either clockwise or counterclockwise direction. In program descriptions and proof arguments, it is convenient to refer to the clockwise (counterclockwise) neighbor of a process using subscript notation: process p_i 's clockwise neighbor is pi+1 and its counterclockwise neighbor is p_{i-1}. The distance from p_i to itself is zero, the clockwise distance from p_i to p_{i+1} is one, and the counterclockwise distance from p_i to p_{i+1} is $n - 1$; the counterclockwise distance from p_i to p_{i-1} is one, and general definitions of distance between p_i and p_j for arbitrary ring locations can be defined inductively. The counterclockwise neighbor of p_i is called the predecessor of p, and the clockwise neighbor is called the successor.

4.1.3.2 Protocol With Known Separation

This section presents a protocol to achieve and maintain a separation of at least C + 1 links between tokens in the unidirectional ring. An implementation of the protocol uses four instantiation parameters, n, m, C, and the choice of which of two programs (delay and relay) are used for nodes in the ring. Only the separation parameter C is used in the protocol, as the domain of a counter, whereas the ring size n and the number of tokens m are unknown for the programs. The separation by C + 1 links cannot be realized for arbitrary n > 1 and m > 1; we require that $m(C + 1) \leq n$ ….(1).

A token in any node can be either resting (denoted by r_i for process i) or enqueued (denoted by q_i for process i). An enqueued token is passed on to the next node in the next round. The protocol consists of two programs: delay and relay. At least one process in the system executes the delay and any processes not running delay run the relay program. The nodes with the counter variable execute the delay program. The counter variable ranges from 0 to C. The variable starts its countdown from C (0, C is the range of the counter), and every time it becomes zero, it restarts the counting from C. A token cannot leave a node if the counter is nonzero. When it is zero, a token is enqueued to be passed on in the next round. In any round, both the relay and delay program starts with accepting an enqueued token from the previous node. The objective of the protocol is to circulate m tokens around the ring so that the distance from one token to the next (clockwise) token exceeds parameter C, and in each round every token moves from its current location to the successor.

A legitimate state for the protocol is a global state predicate, defining constraints on values for variables. To define this predicate, let tokdist denote the minimum, taken over all i such that $r_i + q_i > 0$, of Rdisti (minimum clockwise distance to a token for p_i. Ldisti is for the anticlockwise distance.). The predicate $delay_i$ is true for process p_i running delay and false for the relay processes.

Definition 1. A global state σ is legitimate iff

$\sum_i q_i = m \wedge \sum_i r_i = 0 \wedge (\forall i :: q_i \leq 1)$ ………………………… (2)

$\wedge$ tokdist > C …………………………………………………….(3)

$\wedge (\forall i : delay_i \wedge c_i > 0 \wedge q_i = 0 : Rdist_i = C - c_i)$ …………....(4)

$\wedge (\forall i : delay_i \wedge q_i = 0 : Ldist_i > c_i)$ …………………………..(5)

$\wedge (\forall i : delay_i \wedge q_i = 1 : c_i = C)$ ………………………………(6)

In an initial state, variables may have arbitrary values in their domains, subject to constraint (1). The protocol is self-stabilizing. For proof please refer to our paper [43].

4.1.3.3 Protocol With Unknown Ring Size

Here we consider another design alternative, where the separation between tokens should be maximized, but the ring size is unknown. The technique is straightforward: building upon the delay program, additional variables are added to count the number of rounds needed to circulate a token, that is, the new program calculates n. Two extra assumptions are used for the new protocol: the value of m is known and the number of processes running the delay program is exactly one.

The revised delay program is used here, which introduces $timing_i$, t_i , $ignore_i$, and $ClockBase_i$. The program uses $ClockBase_i$ in place of parameter C, which is periodically recalculated. The method of calculation relies upon knowing m and knowing that all other processes run relay. The program begins a timing phase, which starts a counter t_i at zero, and calculates the number of tokens that are elsewhere in the ring, $ignore_i$. Subsequently, it handles token arrival for purposes of calculating ring size; after $ignore_i$ arriving tokens are ignored, the next token is the one that was released when the timing phase began. Of course, this calculation can be incorrect in early rounds of an execution, but eventually each timing phase culminates in ti having the ring size. With the delay program at one process and relay at all other processes, the system is self-stabilizing to $C = \lfloor n/m \rfloor - 1$. For proof please refer to our paper [43].

4.1.4 DESAL Program To Increase Lifetime Of The Sensors

Figure 16 is the program to calculate average temperature in the sensor motes. Assuming only node zero has the counter, the program is designed based on the separation of circulating tokens paper [43]. The idea is, every node whose id is not zero will record the temperature only if it has a token. Otherwise, it will turn off its temperature sensor. Once a temperature is recorded the node broadcasts the temperature and the base station receives it. Then, the base station calculates the average of the received temperatures. Now, the nodes with idle sensors will also continuously send messages to the base station. But those temperatures will be stale as no temperature recording is being done for sometime. Hence, along with the temperature the recording time is also send to the base in a struct data structure. The values in a struct are always send over the radio together (discussed in detail in chapter 5). Hence, the temperature and recording time will be sent together. The base station then decides depending on the current and received time difference which temperature to accept. The goal of this example is to selectively activate the sensors in the network. This way each node will be sensing temperature for a while and then due to the distance between the tokens it will rest its sensor for sometime. This will enable the node to save energy in the long run. If the node had its sensor active all the time it would deplete of its energy much sooner. In the Figure 16 num(x) is the number of tokens in a node, and pass(x) is the action of passing a token from the host node to the next node clockwise. The implementation of num(x) and pass(x) are discussed after the figure.

There are N processes or nodes and m token rings. Figure 17 shows a system with 4 processes and 3 token rings. In a ring of sensor nodes/motes the objective is to balance the energy consumption of the network to increase its lifetime. To achieve this we assume that the nodes with the tokens will be active sensors in a round, while all the other nodes will have idle sensors waiting for the token. For certain applications this can be power efficient as we have seen in the above example. This will save energy. To achieve this, each token ring should have only one token, and the tokens in the processes should be a certain distance apart. In a faulty initial state, each token ring can have more than one token.

```
component MaxMinTemp
  struct { uint16 temp, uint16 time }  myStruct
    shared struct myStruct myS
    unshared uint16 sum = 0
    unshared uint16 count = 0
    shared   uint16 avg =  0
    unshared uint8 counter = 0 // range is 10
    binding  myStruct mySRemote <- *.MaxMinTemp.myS[20]
    every 1s after 1m
       ID==0:
            foreach n in MySRemote {
            if(($time - n.time) < DIFF) {
            sum = sum + n.temp
                count = count+1 } }
            avg = sum/count
            sum=0
            count=0
   [] (ID==0 && counter>0):
            counter=counter-1
   [] (ID==0 && num(x) > 0 && counter==0):
            pass(x)
           counter=10 // C=10
   []  (ID!=0 && num(x)>0):
           myS.temp =$temp
           myS.time =$time
           pass(x)
```

Figure 16 Program to calculate average temperature in the sensor motes. Each node has at most 20 neighbors. $temp records temperature, whereas $time records local time synchronized with others. In the base (node id =0) if the time difference is within acceptable range, the received temperature will be accepted in the calculation of the average.

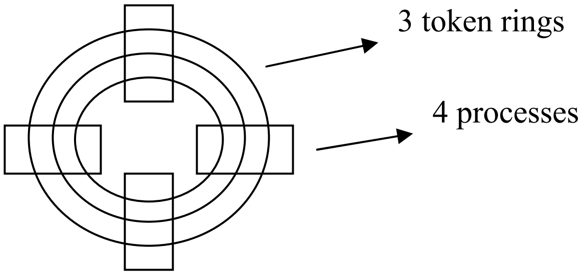


Figure 17 A ring with 3 token rings and 4 processes

The purpose, as explained above, is to eventually self-stabilize the system such that in each round, there is only one token in a single token ring and the tokens are separated by a certain distance. In one round a process can have up to m tokens, as there are m token rings. Our goal here is to ensure that these m tokens are eventually separated by a certain distance in the legitimate state. When a token is present in a process, certain guards of the process are enabled which modify the system state. Shared memory model of computation is considered here. A process i, in addition to reading its own state variable c[i] can also read the state variable c[i-1] of its predecessor process.

We use the idea of Dijkstra's [51] unidirectional token ring self-stabilization algorithm to achieve stabilization where there is only one token per token ring. The algorithm to separate the token by a certain distance is described in the separation of circulating tokens paper [43].

4.1.4.1 Implementation Of Num(x)

c[j] is a state variable for ring j. c[j][i] is a state variable for process i in the j^{th} ring. x represents presence of a token which is denoted by c[j][i] != c[j][i-1]. A token is present, on ring j, at process i, if and only if c[j][i]!=c[j][i-1] (with the exception on node 0), where j is the range of token rings, and i is the range of processes.

For process i, num(x) counts true in

$$c[0][i] \mathrel{!=} c[0][i-1] \vee c[1][i] \mathrel{!=} c[1][i-1] \vee \ldots \vee c[m-1][i] \mathrel{!=} c[m-1][i-1],$$

where, m is the number of tokens and i is process i whose num(x) we are calculating.

4.1.4.2 Implementation Of Pass(x)

The variable c in pass(x) is the same as that in num(x). Consider a variable a $\in$ [0 ... m-1] so that c[a][i] != c[a][i+1]. At anytime a process with a positive number of tokens passes one token to its successor. Now, how can we determine which token to pass if there are more than one in the process? The variable a determines that.

Pass(x) does,

(1) c[a][i] := c[a][i-1]

(2) a=a+j, where j gets to next token such that, c[a][i] != c[a][i+1]

4.1.5 Using RFID Tags In Flume

Three programs are needed to implement the project.

(1) To read water pressure in the flume. There are three RFID antennas containing the tags (RFID tags). The antennas need to be activated one at a time. The recorded pressure will be sent to the base station.

(2) A program to load more marbles. This needs to be done at a specified interval.

(3) The base station with ID=0 will accumulate all the water pressure values and calculate the average.

Since there are multiple DESAL programs running, the binding will have a different format. Instead of binding to the same variable in a process running the same program, here a shared variable will be bound to a different variable in a process running a different program. Figure 18 shows the three programs.

Figure 19 explains how the three DESAL programs work. Antenna.desal is run by the three RFID tages attached to the Antennas in the flume. The tags record the pressure in the flume. Only one of the three tags are active at the time. This is achieved by using a token circulating between these three tags. The base station running AvgFlume.desal program, is reading the shared variable Pressure from Antenna program. After reading all the pressure values from the three RFID tags, the base station is calculating the average pressure. The paper by Nichols [55] deals with a similar problem where a RFID system is implemented to monitor the displacement of coarse particles.

```
      Program (1):
      component Antenna
            shared   uint16 pressure =  0
            shared   token x
            binding uint8 xRemote <-- *.Antenna.x

           every 3s after 50s
              (num(x)>0):
                    $redOn()
                    pressure = $flume_read()
                    $redOff()
                    pass(x)
Program (2):
      component AddMarbles

           every 3s after 200s
                  $blueOn()
                  $flume_add()
                  $blueOff()


Program (3) for ID=0 (Base station):
      component AvgFlume
          unshared  uint16 avg  =  0
          unshared  uint16 sum  =  0
          unshared  uint16 count  =  0
          binding uint16 pressureRemote <- *.Antenna.pressure

          every 8s after 50s
              true:
                foreach n in pressureRemote {
                      sum=sum+n
                      count = count+1
            }
            avg = sum/count
            sum=0
            count=0
            $toggleGreen()
```

Figure 18 The three programs for the RFID Flume case study

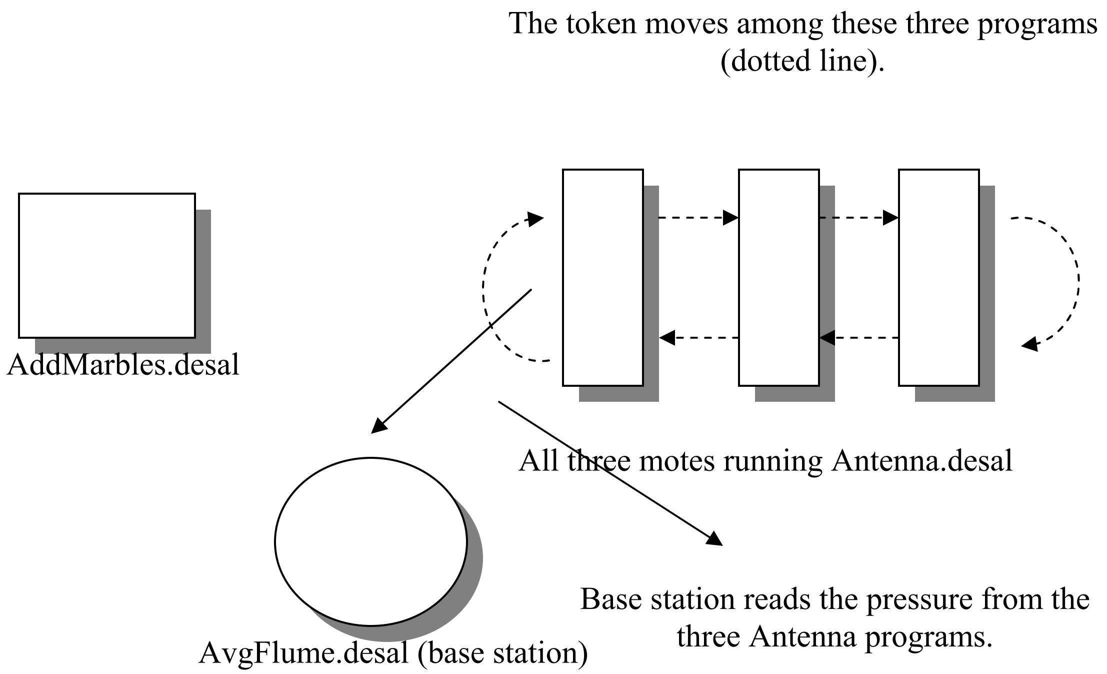

The token moves among these three programs (dotted line).
AddMarbles.desal
All three motes running Antenna.desal
AvgFlume.desal (base station)
Base station reads the pressure from the three Antenna programs.

Figure 19 The working of the DESAL programs for the flume project

CHAPTER 5
DESAL TO JAVA CONVERSION

In this package a DESAL program is converted to an equivalent Java program. A DESAL parse tree is generated as a python dictionary, which is fed into a couple of python programs, which generate the corresponding Java file. The Java file is then compiled with the Java compiler. In the next section we have discussed role of DESAL compiler and explained the grammar file. After that we have given an example of a DESAL program converted to an equivalent Java program. The aim behind this work is to make DESAL compatible with the Java platform. This is a unique contribution of DESAL. By doing this Java program can communicate with the sensor nodes via SerialForwarder. This way we no longer have to depend on NesC and TinyOS for the communication.

5.1 DESAL Compiler And Grammar

The DESAL Compiler is written in Python version 2, using a parser module provided by the Dparser project (hence, both Python 2 and Dparser need to be installed to implement the DESAL Compiler). To convert DESAL program to NesC, our compiler generates a python dictionary after parsing, which is fed into the Clemson's semantic checker. If there is no semantic error the DESAL program is converted to equivalent NesC code. To validate the correctness of our grammar rules, we have used Jython to feed the python dictionary into the semantic checker written in Java. Clemson has already verified the correctness of their compiler by converting the DESAL program to corresponding NesC code, which successfully ran on the motes giving the desired result. Therefore, replacing the Clemson compiler with ours and translating a DESAL program to equivalent correct NesC code validates the correctness of our compiler.

5.1.1 Compiler Components

The grammar is specified in a file grammar.def. The program makeDparser.py converts grammar.def to the Dparser style of grammar specification. Constructing the AbstractSyntaxTree is done by invoking Dparser with the source of a DESAL program; as a result a tree (Python dictionary) representing the parsed input is created. The module Traverse.py is a central tool for analysis of the parse tree and generating code based on the tree. There is a "swap chains" function that transforms certain structures in the AbstractSyntaxTree. Codegen.py creates the objects (data architecture) of code generation. Java code is generated by CodeAssign.py. The codes generated are filled in Skeleton.java to create the final java file called Proto.java.

5.1.2 DESAL Grammar

The formal grammar rules of DESAL are specified in the grammar.def file.

5.1.2.1 Explanation Of Grammar.def

The entire file (omitting trivial rules) is given in Appendix B. We invented a small, abbreviated syntax to generate dparser grammar, which is a style of commented Python. Here, the abbreviated syntax has two basic forms, single rules and multiple rulesets. Each rule or ruleset starts with a string, in column 1, looking like #nn#, where 'nn' is a number. The meaning of this number is taken from the Java Class numbering given by Dalton and Hallstrom, in their original Java compiler. An example is the following rule:

```
#2#
  componentDec: "component" var_id subComponentListNull
      if type(term2) == types.ListType:
            term2=term2[0]
    node["var_id"] = term1
    node["subComponentListNull"] = term2
    node["name"] = "component"
```

Notice that the rule starts with #2#, which means this will generate a dparser pattern (the first line following the #2# is the pattern), which eventually will generate a node of type 2, when a DESAL program is compiled into an Abstract Syntax Tree. The remaining lines in the rule are Python statements executed after dparser matches to the pattern. Here, some conventions are:

1. numTerms is a Python local variable equal to the number of terms matched by the dparser pattern. This may be variable, because dparser rules can have optional matching expressions (so numTerms isn't always the same number).

2. term0, term1, term2, etc, refer to the terms matched by the dparser pattern. You can refer to these terms in string manipulation and comparison code, but sometimes the terms are not Strings, but are lists (this depends on how your grammar is defined). Notice above, the assignment: term2=term2[0]. This assignment is based on the assumption that term2 is a list prior to the assignment, and the first item of the list replaces local variable term2 (of course, this assumption is valid because of the "if" test on term2's type!).

3. The "output" of the rule is always a Python dictionary called, locally, 'node'. Here, you can add particular key/value things to this dictionary. Notice that above, the key "var_id" is added to the node. Some keys are standard, and be careful about these:

1. node["CaseNo"] -- automatically assigned, this will be the node's number (for the example above, it is 2).

2. node["LineNo"] -- automatically assigned, this is the line number of the DESAL program source for the matching program fragment.

3. node["ColumnNo"] -- like LineNo, but for column number.

4. node["Name"] -- optional; used basically for debugging and pretty printing of the parse tree by some tools.

For some cases, one dparser pattern could possible generate different node numbers, depending on inputs. To allow this, we have rulesets. A ruleset starts, like a rule, with a #xxx#-string in column 1, but the 'xxx' here will be a comma-separated list of numbers; these are the possible node numbers for the ruleset. Example:

```
#3,4#
  subComponentListNull: subComponentList?
     if not term0:
          #4#
     else:
        if type(term0) == types.ListType: term0=term0[0]
        #3#
        node["subComponentList"] = term0
```

This example shows a ruleset for node types 3 and 4. Notice that we see Python code interwoven with #3# and #4#, which are indicators of the specific definitions for node types 3 and 4. In the example, node type 4 has no special dictionary key/value items added, whereas type 3 has one key/value item added. Our tool validates that the lines following a ruleset definition contain entries for all the rules that should be defined. Thus, following #3,4#, there has to be some line #3# and some line #4# (and, of course, no line #5# or other crazy numbers).

If you look at the Dparser specification of the grammer, you see a mix of productions (rules to parse) and Python code. Associated with each rule is a snippet of code that initializes a tree node when the rule executes. For example, consider the following rule:

```
def d_Guard(t, s, nodes, this):
 '''Guard: expr ':' stmntListNull '''
```

```
global term0, term1, term2, term3, term4, term5, term6, term7, term8, term9
numTerms = len(t)
assignTerms(t)
node = dict()
node["LineNo"] = nodes[0].start_loc.line-1
node["ColumnNo"] = nodes[0].start_loc.col+1
node["CaseNo"] = 29
if type(term2) == types.ListType:
    term2=term0[0]
node["expr"] = term0
node["stmntListNull"] = term2
node["name"] = "Guard"
return node
```

Intuitively, this production specifies that a guard statement is an expression followed by a colon followed by a statement list. The Python code associated with this rule creates a dictionary and returns that to Dparser, which is driving the parsing process. Some entries in the new dictionary are standard keys for all nodes:

LineNo identifies the source code line number (useful for generating messages about the Desal program source later during code analysis or code production stages). *ColumnNo* identifies the column number where the guard statement begins. *CaseNo* is used to identify the type of node; this is crucial in later stages that analyze the tree, so it is simple to know what type of node and which dictionary keys and values it contains; hence, a node with CaseNo of 29 is a guard statement node. Name is for documentation purposes (we sometimes made a pretty formatted picture of a parse tree).

Other keys of the dictionary depend on the particular rule and node type. Thus, to make sense of parsing and code analysis/production, you need to arrange that the rule with associated Python snippet have what's needed when the tree is later analyzed.

A node is a dictionary with a key that consists of the string "CaseNo". A child of a node is this: any key in a dictionary with an associated value that is also a dictionary is a child node. Thus if X is a node, and type(X["abc"]) == dict, then X has a child (which is associated with key "abc").

5.1.2.2 Traverse.py

Traverse(V,Root,FilterDict) is a general higher-order function to traverse the nodes of a tree recursively. Traverse is designed to be the one template to satisfy all tree traversals for any reason; it can be either bottom-up or top-down in its processing, or even a mixture of these depending on node type (yes, this probably goes further than needed in its generalizing). The three arguments to Traverse are:

- V is the code generation object.
- Root is the tree root (or could even be any node) of the AbstractSyntaxTree.
- FilterDict is a dictionary with entries of the form:

43

(M,c), where 43 is a CaseNo for a node that should be processed; M is an object that has a method called mutate that should be applied to a node of CaseNo 43. And c is a character, either 'b', 'p' -- which stand respectively for pre-order, or post-order.

-1

(M,c) is a wildcard that matches any nodetype.

The idea is that FilterDict is essentially a list of things to do for selected nodes. You can have a FilterDict for only some types of nodes, selected by the node CaseNo, or you can have a wildcard selector. The mutate method is invoked on the node (we make it a method so that M can have enough state for the mutate invocation to have a rich history and environment, for general programming purposes). A mutate invocation could change the node (remember, the node is a dictionary) by adding keys, assigning to current keys, or mutate might make no change but instead accumulate information in some component of the object M, such as a list.

Traverse will look at every node in the tree, where a node is anything that has type 'dict' (dictionary in Python) and has a CaseNo attribute/key (other things are ignored). For each node inspected, Traverse will skip over the node if its CaseNo isn't in the FilterDict. For a node T that is in the FilterDict, Traverse will invoke M.mutate(V,T). Depending on whether c is 'b' or 'p', the invocation M.mutate(V,T) will be done before or after (respectively) Traverse recursively handles all children of T.

```
import Mobject  # for access to methods applied to nodes
import Codegen  # for access to the code generator object
def traverse(V,T,FilterDict):
  if type(T) != dict:  return
  if "CaseNo" in T and T["CaseNo"] in FilterDict:
    # found a node to process, but only do it now if
    # the modality is pre-order traversal for this FilterDict entry
    (M,c) = FilterDict[T["CaseNo"]]
    if c=='b':  M.mutate(V,T)
  if -1 in FilterDict:
    (M,c) = FilterDict[-1]
    if c=='b':  M.mutate(V,T)
  #  Part 2:  recursively take care of T's children
  for e in T:  traverse(V,T[e],FilterDict)
  #  Part 3:  do any post-order processing of the node
  if "CaseNo" in T and T["CaseNo"] in FilterDict:
```

```
# found a node to process, but only do it how if
# the modality is post-order traversal for this FilterDict entry
(M,c) = FilterDict[T["CaseNo"]]
if c=='p':  M.mutate(V,T)
```

Notice above that the wildcard is only implemented for cases of pre-order application of mutate. This may be a bug or may be a feature which has not been exercised. Also, notice that the wildcard does not test whether CaseNo is a key; this might also be a bug.

5.1.2.3 SwapChains.py

The tree that Dparser produces isn't quite suited to some analysis and code generation. The problem turns out to be that some "linear chains" of nodes in the tree are backward for our purposes of tree traversal. To deal with this, we have a special transformation of the tree which reorders such chains. The module SwapChains.py does the chain reversal, using a kind of mark-and-sweep programming style (see wikipedia for references on this technique).

```
import sys
import Traverse
import Mobject
class remomark(Mobject.Mobject):
  def mutate(self,v,t):
    if "swapchain" in t:  del t["swapchain"]
def unmark(V,T):
Traverse.traverse(V,T, { -1:(remomark(),'b') } )
```

The class shown in Figure 20 gives us an object with a mutate method, suitable for using Traverse. The unmark function then invokes Traverse, passing a remomark object. Notice that all that the mutate method does here is to remove any key named "swapchain" from every node. The unmark function is used later, after some marking and swapping has been done, essentially cleaning up things at the end of swapping chains.

Above, you see that class swap is a specialized object (initialized with a CaseNo and the names of two types of node). The mutate method enumerates a subtree, reverses its order, and marks a chain. The actual source code in the Figure 20 has more informative comments.

From the swap method in figure 21, we see that "swapchains" is just a sequence of chain reversals, each using the swap object and its mutate method to reverse a particular type of chain.

```
        class swap(Mobject.Mobject):
        def __init__(self,RecurseType,ElemKey,ChainKey):
           self.RecurseType = RecurseType  # e.g., 32 for stmntList
           self.ElemKey = ElemKey          # e.g., "stmnt"
           self.ChainKey = ChainKey        # e.g., "stmntList"
        def mutate(self,v,t):
           if "swapchain" in t: return
           chain = [ t ]
           while True:
              last = chain[-1]
              if last[self.ChainKey]["CaseNo"] != self.RecurseType:  break
              chain.append(last[self.ChainKey]
           termChain = [ ]
           for c in chain: termChain.append(c[self.ElemKey])
           termChain.append( chain[-1][self.ChainKey][self.ElemKey] )
           for c in chain:
               c[self.ElemKey] = termChain.pop()
           chain[-1][self.ChainKey][self.ElemKey] = termChain.pop()
           for c in chain: c["swapchain"] = True
ERROR: EOF in multi-line statement
```

Figure 20 Swap class in SwapChains.py

5.1.2.4 Objects For Compiling

To produce code from the AbstractSyntaxTree, several conceptual objects are defined in Codegen.py:

The Body object represents the body statement information in a Desal program, such as evaluation frequency, period, and a list of guarded statements contained in the body.

BindingSpec object represents one binding specification, which has a type, target, source, and possibly a range for indexing in a node's neighborhood. Variable represents a declared variable: it has a name, type, and later is given binding and numeric codes.

For a "struct" declaration, a Struct object is defined with things like size, name, values, and so on. Each function declaration has fields for name, parameters, and statements in the function's body. Scopes are objects used during AbstractSyntaxTree analysis to have a context for nested "foreach" statements. Generated code for the foreach construct introduces an implicit loop variable, which has to be different from the variables used in nested foreach statements (hence, scoping is important). Finally, there is an object which handles all of the above and more, so that recursive processing of an AbstractSyntaxTree or subtree has access to variables, statements, bindings, and so on, represented by the objects described above. This is called CodeObject. The CodeObject has fields to contain information for all tables to be generated for the DESAL program; it also can collect generated code, counters used to build temporary names, and so on.

In addition to these objects, Codegen.py also defines enumerated constants for variable types: uint8, uint16, uint32, bool, and struct. The basic theme for building the objects described above is to use Traverse.py on the AbstractSyntaxTree with appropriate filters, passing some handle to the CodeObject so that the various objects can be created and saved. The processing of the AbstractSyntaxTree for code production is preceded by some setup phases.

```
def swapchain(V,T,ChainType,ElemKey,ChainKey):
   unmark(V,T)    # unmark the tree initially
   Traverse.traverse(V,T, { ChainType:(swap(ChainType,ElemKey,ChainKey),'b') } )
   unmark(V,T)    # unmark the tree afterwards
def swapchains(V,T):
   # Problem setting:  each of the following grammar rules generates a tree with
   # chains of nodes that are _backward_ from what the source DESAL input has
   #
   #   CaseNo: 5, 6 - '''subComponentList: subComponentList subComponent | subComponent '''
   #
   #   CaseNo 9-14 - '''DecList:  DecList structDec | DecList stateDec | DecList bindingDec | structDec | stateDec | bindingDec '''
   #
   #   CaseNo 17, 18 - '''bindingVarList: bindingVarList ',' bindingVar | bindingVar '''
   #
   #   CaseNo 27, 28 - '''GuardList: GuardList '[' ']' Guard | Guard '''
   #
   #   CaseNo 32 -34 - '''stmntList : stmntList stmnt | stmnt | "error" '''
   #
   #   CaseNo 46, 47 - '''ElseIfList : ElseIfList ElseIf | ElseIf '''
   #
   #   CaseNo 86, 87 - '''exprList : exprList ',' expr | expr '''
   #
   #   CaseNo 95, 96 - '''varDecList: varDecList varDec | varDec '''
   #
   #   CaseNo 106, 107 - '''varList: varList ',' var | var’’
   #
   #   The swapchains() method mutates the tree so that these nodes are in an order
   #   that matches our intuitive understanding of the DESAL program text.
   #
   #   NOTE:  currently, only these cases are handled;  but it is easy to generalize
   #          32 - stmntList
   #          27 - GuardList
   #           5 - subComponentList
   #         ( other cases not yet handled )
   swapchain(V,T,32,"stmnt","stmntList")
   swapchain(V,T,27,"Guard","GuardList")
   swapchain(V,T,5, "subComponent","subComponentList")
```

Figure 21 Swap method in SwapChains.py used to reverse a particular list.

This counts the number of variables, reverse some paths in the tree, and so on. The subsections below talk about modules that do this initial processing.

Setvars.py module has the objects and functions needed to set up variable tables. The main function is setupVariables(V,T), where V is the CodeObject and T is a (subtree) of the AbstractSyntaxTree. An invocation of setupVariables uses Traverse.py recursively in several ways with mutate methods that set up fields within the CodeObject: namely, VList, VarCount, Vars, StructList, StructInitList.

Setbinds.py module establishes the binding tables. This module contains the setupBindings(V,T) function, which should be invoked soon after the AST and Code Generation objects are constructed. An invocation of bindProcMobject uses Traverse.py recursively with mutate methods that iterate through all the binding variables declarations. This extract the variable names, binding scope, component name to which it is associated, shared variable name to which it is associated and target range.

Setstruct.py module contains the setupStruct(V,T) function, which should be invoked soon after the AST and Code Generation objects are constructed. It will establish struct tables. An invocation of structProcMobject uses Traverse.py recursively with mutate methods that iterate through all the structure declarations. This extracts the structure names, the data fields of the struct and the size of the struct.

Setfunc.py module contains the setupFunction(V,T) function, which should be invoked soon after the AST and Code Generation objects are constructed. It will establish function tables. An invocation of funcProcMobject uses Traverse.py recursively with mutate methods that iterate through all the function declarations. This extracts the function names, the parameters of the function, the statements in the function body and the return type of the function.

5.1.2.5 Generating Java Code

The python file to generate the Java code from the AbstractSyntaxTree is CodeAssign.py. This module assigns Java code attributes to nodes of the AbstractSyntaxTree,that is, it makes the assignment

```
node['java'] = ... some string containing Java ...
```

The module contains different classes for different data structures and operations. Each class contains a mutate() method which is called in Traverse.py to convert each DESAL statement to the equivalent Java statement. Mobject is imported to provide access to methods applied to nodes. Mobject is a fake class, provided for traversing the AST. An instance of Mobject will provide a 'mutate' method, which is invoked on each relevant node of the AST during traversal to add fields or change fields of AST nodes, and possibly change the CodeObject (repository of state during the code generation phase).

CodeAssign.py creates different classes to do the code conversion from DESAL to equivalent Java code. Before accessing any class methods, all the DESAL variables are converted to the objects of a class called DV. Then onwards all the operations on the variables are done with DV methods. E.g. for adding two DESAL variables a and b, the equivalent Java code will be `DV.Add(a, b),` where a and b are now DV objects.

Below we show some example of code conversion by referring to the classes used.

5.1.2.5.1 DESAL Code

```
1. foreach n in nmax {
 2.      if (n > max) {
 4.            max = n
```

```
       }
}
```

5.1.2.5.2 Equivalent Java Code

```
for(int n=1; n<num_of_neighbors; n++) {
    DV.push(DV.Gt(nmax[i], max))
    if(DV.Boolean(Dv.pop)) {
      DV.set(max, nmax[i])
     }
}
```

The *ForEach()* class reads in the scope of the binding variable from the for loop, which is n here. If a shared variable in DESAL is bound to N processes, in the equivalent Java code an array of size N is created, where the name of the array is the name of the remote variable. E.g. if the binding is, `nmax <- *.funcTest.max,` where * denotes the N remote process, in Java an array named nmax is created. In DESAL the line `for n in nmax` n denotes each remote process each time the loop is entered. In the equivalent Java code, a for loop is created with n as the scope of the loop, and it goes through the nmax array values.

boolDyad (Dyad) class converts Boolean operation to DV method calls, which does the Boolean operation in the equivalent Java code. The Dyad() class contains a method called exprEval() which does the code conversion. E.g. `n > max` in DESAL code will be `DV.Gt(n, max)`, where Gt stands for the Greater Than function.

The *ifS()* class converts a DESAL if statement to an equivalent Java if statement. E.g. `if (n > max) {`will become `DV.push(DV.Gt(nmax[i], max)) if(DV.Boolean(Dv.pop)) {`.

AssignS() class converts an assignment expression to the equivalent DV method call. e.g. `max = n` becomes `DV.set(max, n).`

pushEval() class: When there is an expression in parenthesis, the logic is to generate a Java push on the result of the expression. E.g. is we have, `(expression),` then the equivalent Java code will be `DV.push(expression).` The expression refers to a DV function call. The push is done because in a long expression with multiple parentheses, an expression is evaluated by solving the innermost parenthesis expression first. The expression will be then popped out into the longer expression. E.g. `if (n > max)` is converted to `DV.push(DV.Gt(nmax[i], max)) if(DV.Boolean(Dv.pop)) {.`

Struct is a new data structure introduced in DESAL. In a struct we can group more than one element. E.g. suppose we have two variables to store temperatures. Say, one variable stores the maximum temperature and the other one stores the minimum temperature among the min and max temperatures recorded from neighbors. These two variable record different temperatures, but they are similar in functionality. It makes sense to have these two variables in a group. This group forms a struct. Figure 22 shows the program for calculating minimum and maximum temperatures using struct. It is important that we send the members of a structure together in a message. This is because, since they are similar kind of variables, another node should read struct member variables with the same timestamp. Otherwise, two values send at different times may not be useful. That’s why in DESAL we don’t allow struct members to get separated into different messages. This also means, struct variables cannot be bound individually. They need to be bound all together. In the above example, if the max and the min temperatures are read in different times, the min and max could have been recorded in different times. If one message payload is not long enough to accommodate the struct, we send the struct in the next payload. Hence, one limitation of struct is, its size should be less than or equal to the message payload size. But having the struct members together in one payload is more important to maintain the relevance of the values.

Setstruct() class converts a DESAL struct to an equivalent Java class. E.g. struct { uint8 v1, uint16 v2 } myStruct will be converted to,

```
public static class structClass {
}
public static class myStruct extends structClass {
  public DV v1 = new DV(0, "unshared", 0);
  public DV v2 = new DV(1, "unshared", 0);
 public myStruct ( DV v1copy, DV v2copy) {
 v1 = v1copy;
 v2 = v2copy
 }}
```

A super class named structClass is created from which all the structure classes are extended. A constructor in the structure class, here myStruct, initializes the value of the structure member variables. Another unique feature of DESAL is the inclusion of *function* in the language. A function in DESAL is declared in the program component before the bodies. We need a function to group together commands that can be repetitive.

Or, to make the program more readable, we can move a bunch of inline codes to the function and then call the function instead of writing the multiple lines of code in the main body. The unique feature of function is the code inside uses only global variables. No new local variable is declared. This can significantly reduce the stack overhead of the program, thus saving memory and running time. When a function is called in the body, the current state of the body is pushed into the stack and the control transfers to the function subroutine. This stack is pretty small. It only stores the main program and the function information. No new stack is created to store the function local variables. This can significantly reduce the stack overhead.

After the execution of the function is done, the control returns to the main body where it has left off and pops out the state variables of the body stored before the function call. Let us consider an application where we calculate neighborhood average of average temperatures from the neighbors at two different time frequencies. Hence, we need to calculate the same average in two different bodies. So, we need to use the average function twice. Instead of writing inline code, we call a function that will do the work. The program is shown in Figure 23.

```
component MaxMinTemp
          struct { uint16 min, uint16 max }  myStruct
          shared struct myStruct myS
          binding myStruct mySRemote <- *.MaxMinTemp.myS[20]

          every 5s after 50s
            true:
                  MyS.max=$temp()
                  MyS.min=$temp()
                  foreach n in mySRemote {
                        if (n.max > MyS.max) {
                              MyS.max = n.max
                        }
                         if (n.min < MyS.min) {
                              MyS.min = n.min
                         }
                  }
```

Figure 22 DESAL program to calculate min and max of neighborhood temperatures. Each node is connected to 20 neighbors. Initially, local temperature is assigned to min and max. Then, the recorded min and max are compared to the min and max from the neighbors to calculate the neighborhood min and max. Here we can see that if the min and max from the neighbors are received at a different time, the calculation will give us wrong result, as we want to calculate min and max at the same time.

A function in DESAL is converted to an equivalent Java function by the method in *funcCall()* class. E.g. DESAL function,

```
uint8 test(){
```

```
sum=sum+$temp()

count=count+1

return count}
```

is converted to a Java function as,

```
public static DV test(){

    DV.set(sum,DV.add(sum,Runf.temp( )));

    DV.set(count,DV.add(count,1));

    return count;

}
```

```
component AvgTemp
  unshared uint16 sum = 0
  unshared uint16 count = 0
  shared   uint16 avg =  0
  binding  uint16 avgRemote <-*.AvgTemp.avg[20]
  uint16 average ()
       {
        sum=0
        count=0
        foreach n in avgRemote {
            sum = sum + n
                count = count+1
        }
        return sum/count
       }

  every 1s after 1m
    true:
         avg = average()

  every 1s after 2m
    true:
        avg = average()
```

Figure 23 This program calculates average temperature of averages from the neighbors twice at different time frequency. Instead of writing the same code for doing average two times, the program calls the average() function in the two bodies. This demonstrates how a function can make a program shorter and easy to read.

genJava() and *exprList()* classes create a filter (an array) of all the functions from the different classes including the ones mentioned above to do the code conversion. In the exprList() class the functions in the filter are executed by the Traverse() function in Traverse.py. since, all the function calls for the different expressions in DESAL program are sent to Traverse(), Java code for all the DESAL expression are created by the exprList() class' method. In genJava() the program is broken apart into variable declarations, bodies, and main sections.

In *BodyPrep()* class, the body in DESAL is broken into guards and statements in pairs which are converted to the equivalent Java expressions as an if-then structure.

In *Copy()* class, the converted Java codes are copied to a Java file called Skeleton.java. Skeleton.java contains the Java codes, which are common to all the Java files created from DESAL codes. E.g. the DV class, the Runf class for system variables, and so on are common to all the created Java classes. Copy() class fills in the gaps by inserting the Java code for a particular DESAL program to create the equivalent Java program.

5.2 DESAL Application In Java

In this section we will see a DESAL program and its equivalent Java program. Below is the program Blink in DESAL.

```
component Blink
    every 3s after 0s
        true:
            $redOn()
    every 3s after 1s
        true:
            $redOff()
```

Equivalent Java Program is given in Appendix A.

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

Previous research works (discussed in the literature review) have investigated many features like time synchronization, neighborhood management, high level programming interface etc, essential for developing a high-level user-friendly programming language for wireless sensor networks. DESAL attempts to combine all these features to present a simple high-level sensor network programming language. The integrated middleware hides all the low-level detail from the programmer.

One unique feature of DESAL is its state based programming property. The programs are written in guarded-action format. Therefore, there is no hidden context like event or interrupts. The middleware services are automatically integrated in a program during compilation. The middleware offers services, such as, time synchronization, neighborhood management, dynamic binding and message communication. The static construction of these low-level features can enable efficient usage of data structures and expert programming to exploit the advantages of NesC and TinyOS. This can result in efficient memory utilization and economic power usage. DESAL can also be converted to Java, where the computer interacts with the sensor motes via the SerialForwarder.

DESAL introduces a variable type called 'token'. The idea is, when a process has a token it performs certain action and changes its state, while others wait for their turn. One of the case studies we have done uses token variables to achieve energy efficiency by separating the tokens in a ring by a certain distance. This has been discussed in detail in our paper Separation of Circulating Tokens [43]. Another case study with tokens involves selective activation of an RFID tags in a network.

Struct is a new data structure introduced in DESAL. In a struct we can group more than one element. It is important that we send the members of a structure together in a message. This is because, since they are similar kind of variables, another node should read struct member variables at the same time. Otherwise, two values send at different times may not be useful. That's why in DESAL we don't allow struct members to get separated into different messages. This way struct can be handy when it comes to dealing with similar kinds of variables. The functioning of the struct has been illustrated with a case study.

Function is also newly introduced in DESAL. The unique feature of function is the code inside uses only global variables. No new local variable is declared. This can significantly reduce the stack overhead of the program, thus saving memory and running time. Function has been illustrated with a case study.

Most of the research work shows that the program running in the base station is usually different from the program running on the sensor motes. Usually, high level language like Java runs in the base, while low level language like NesC is run on the motes. Since, DESAL can be converted to both NesC and Java, DESAL codes can be written for both the base station and the motes. There is no need to write programs in separate programming languages for them. This is an important contribution.

6.2 Future Work

The following improvements to DESAL have been proposed.

DESAL Extensions: The DESAL development model is not well suited to low-level programming task or tasks involving hard real time processing constraints. In the future, developers must be able to introduce new services using low-level primitives and be able to access those services from DESAL guards and actions [42].

Multi-Hop bindings: A multi-hop binding implementation would enable a base station to establish a multi-binding to each cluster-head in the network, where all the cluster heads may not be a neighbor [42].

Running multiple programs: As discussed in the RFID flume case study, we need three different DESAL programs to run in the same network. At present DESAL cannot allow that. Currently DESAL can run identical programs in the sensor nodes in a network. In the future we could enhance DESAL to allow different programs to run in the same network.

Token in Java: We have introduced variable of type token in DESAL. But token hasn't been implemented in the Java program converted from a DESAL program. This can be done in the future.

For the separation of circulating tokens an interesting question is whether there can be a hybrid or uniform protocol when the ring size and the separation constant are unknown. For the style of algorithm proposed in our paper [43] for unknown ring size, we conjecture the answer is negative. If one delay process pi has an accurate estimate for maximum separation $d = c_i + 1$ and does not delay any arriving token, another process p_j may have either a larger, inaccurate estimate, or may perceive that tokens are unaligned with its counter and therefore delay some arriving tokens. Such delay would lead to p_i detecting an apparently larger ring size, since the measured traversal time around the ring would include p_j 's delays. Hence pi would raise its estimate for the separation value. Note that the problem may admit other types of algorithms: for example, if tokens are allowed to carry data, this would enable processes to communicate. Whether such increased communication power is useful is an open question. Another direction would be to use randomized timing, so that different delay processes do not interfere. An obvious direction for future research is to move beyond rings to other topologies. Figure 24 below suggests how a virtual ring can be mapped upon a network, which could provide separated token circulation. Another possibility is to map distinct rings upon a network to cover all nodes and attempt to coordinate the timing of token circulation in these rings where they intersect [43].

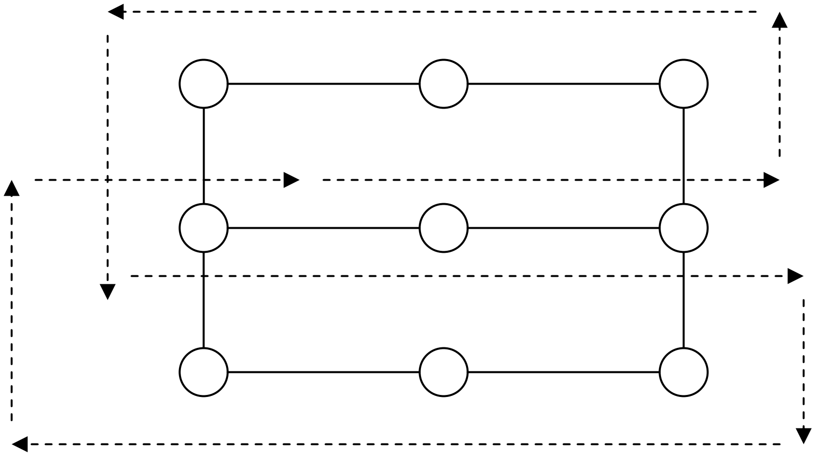


Figure 24 Virtual ring can be mapped upon a network, which could provide separated token circulation.

APPENDIX A

EQUIVALENT JAVA PROGRAM FOR BLINK.DESAL

The figure A1 shows equivalent java program for Blink.desal. The figure starts in the next page.

```
       // import tinyos message to enable message communication with the base
       // station via the SerialForwarder
       import java.util.*;
       import java.lang.Thread;
       import java.lang.Object;
       import java.util.concurrent.TimeUnit;
       import java.lang.System;
       import net.tinyos.message.*;
       import net.tinyos.util.*;
       import java.io.*;

       // the name of the java file is Proto.java
       public class Proto extends net.tinyos.message.Message implements
MessageListener
       {
       // No variables declared for the Blink application
       // ID is a constant giving the identity code for a sensor node
             static DV ID = new DV();
             //Declare shared and unshared variables public static
             public static class structClass {
             }
       // class RunF Declare System Variables
             public static class Runf {

                    public static DV moteId() {
                           DV val = new DV(0, "shared", 0);
                           return val;
                    }
                    public static DV tsr() {
                           DV val = new DV(0, "unshared", 0);
                           return val;
                    }
                    public static DV temp(){
                           DV val = new DV(0, "unshared", 0);
                           return val;
                    }
       // redON() and redOff() will be called from the two bodies as per the
          code: very 3s after 0s
                      true:
                          $redOn()

                 every 3s after 1s
                      true:
                          $redOff()
                    public static DV redOn(){
                           DV val = new DV(0, "unshared", 0);
                           return val;
                    }
                    public static DV redOff(){
                           DV val = new DV(0, "unshared", 0);
                           return val;
                    }
                    public static DV blueOn(){
                           DV val = new DV(0, "unshared", 0);
                           return val;}
                    public static DV blueOff(){
                           DV val = new DV(0, "unshared", 0);
                           return val;}
```

Figure A1 Java program Proto.java equivalent to Blink.desal

```
             public static DV greenOn(){
                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV greenOff(){
                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV toggleRed(){
                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV toggleBlue(){
                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV toggleGreen(){
                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV source(DV a){

                   DV val = new DV(0, "unshared", 0);
                   return val;
             }
             public static DV bound(DV a){

                   DV val = new DV(3, "unshared", 0);
                   return val;
             }
             public static DV age(DV a){

                   DV val = new DV(0, "unshared", 0);
                   return val;
             }

       }

       //Declare Bodies

      // first Body: every 3s after 0s
public static class Body0 implements Runnable {
             public static DV DV = new DV();
             public static Runf Runf = new Runf();
             private int frequency = 3 ;
             private String frequencyUnit = "sec" ;
             private int period = 0 ;
             private String periodUnit = "sec" ;

             private long freq = 0 ;

             private long perd = 0 ;

             public void run() {
              while(true) {
                   int num_guards = 1 ;
                   long starttime = System.currentTimeMillis();
                   int gindex =0; // guard index
```

Figure A1 continued

```
                           if(frequencyUnit.equals("sec")) {freq =
frequency*1000; }
                           if(frequencyUnit.equals("min")) {freq =
frequency*60*1000; }
                           if(frequencyUnit.equals("hour")) {freq =
frequency*60*60*1000; }
                           while (starttime < freq) {
                                  switch (gindex) {

       // true: $redOn()


             case 0:
                                         if(DV.getBoolean(DV.ConstBool(true))){
                                               Runf.redOn( );
                                                }
                                  default: System.out.println("invalid guard
number");
                                   }
                           if (gindex < num_guards) {gindex ++ ;} else {gindex
=0 ;}
                           starttime = System.currentTimeMillis() - starttime;
                           }
                           if( periodUnit.equals("sec")) {perd = frequency*1000;
}
                           if( periodUnit.equals("min")) {perd =
frequency*60*1000; }
                           if( periodUnit.equals("hour")) {perd =
frequency*60*60*1000; }
                           try{ Thread.sleep(period*perd); } catch
(InterruptedException e) {}
                     }
              }
              }
       // second body: every 3s after 1s
       public static class Body1 implements Runnable {
                    public static DV DV = new DV();
                    public static Runf Runf = new Runf();
                    private int frequency = 3 ;
                    private String frequencyUnit = "sec" ;
                    private int period = 1 ;
                    private String periodUnit = "sec" ;

                    private long freq = 0 ;

                    private long perd = 0 ;

                    public void run() {
                     while(true) {
                           int num_guards = 1 ;
                           long starttime = System.currentTimeMillis();
                           int gindex =0; // guard index
                           if(frequencyUnit.equals("sec")) {freq =
frequency*1000; }
                           if(frequencyUnit.equals("min")) {freq =
frequency*60*1000; }
                           if(frequencyUnit.equals("hour")) {freq =
frequency*60*60*1000; }
```

Figure A1 continued

```
                           while (starttime < freq) {

                                 switch (gindex) {
       //true: $redOff()

             case 0:
                                        if(DV.getBoolean(DV.ConstBool(true))){
                                               Runf.redOff( );
                                                }
                                 default: System.out.println("invalid guard
number");
                                  }
                           if (gindex < num_guards) {gindex ++ ;} else {gindex
=0 ;}
                           starttime = System.currentTimeMillis() - starttime;
                           }
                           if( periodUnit.equals("sec")) {perd = frequency*1000;
}
                           if( periodUnit.equals("min")) {perd =
frequency*60*1000; }
                           if( periodUnit.equals("hour")) {perd =
frequency*60*60*1000; }
                           try{ Thread.sleep(period*perd); } catch
(InterruptedException e) {}
                     }
              }
              }

             // Declare Message

                    /** The default size of this message type in bytes. */
                 //public static final int DEFAULT_MESSAGE_SIZE = 28;

                 /** The Active Message type associated with this message. */
                 public static final int AM_TYPE = 147;

                 /** Create a new ProtoMsg of size (variable). */

             public Proto() {}

             public Proto(int size){
                     super(size);
                    amTypeSet(AM_TYPE);
                     }

             // Declare Message Communication

                     MoteIF mote;
                    void run() {

                       mote = new MoteIF(PrintStreamMessenger.err);

                    }
                    public synchronized void messageReceived(int dest_addr,
Message msg)
                    {

                   }
```

Figure A1 continued

```
                    // send data to the SerialForwarder
                    void sendMsg() {



                 System.out.println("message sent");

             }


             // DESAL Methods

             public static class DV{
                    private int vartype; // uint, bool, structtype
                    private boolean shared = false, unshared = false, constant
= false;
                    private long varvalue; // actual value
                    private boolean varvaluebool;
                    private String sourceId; // declare binding type and
parameters
                    private DV sourceVar;
                    private int targetRange;
                    private boolean bound;
                    private int mote_id;
                    private structClass DVStructVal;
                    private static Stack<DV> valD = new Stack<DV>();

                    public DV() {}

                    public DV(int vtype, String scope, long value) { // for
local variables
                    varvalue = 0; // default
                    vartype = vtype;
                    if (scope.equals("shared")) { shared = true; }
                    if (scope.equals("unshared")) { unshared = true; }
                    if (scope.equals("const")) { constant = true; }
                    varvalue = type(vtype, value);
                    }

                    public DV(int vtype, String scope, boolean value) { // for
local variables
                    varvaluebool = false; // default
                    vartype = vtype;
                    if (scope.equals("shared")) { shared = true; }
                    if (scope.equals("unshared")) { unshared = true; }
                    if (scope.equals("const")) { constant = true; }
                    varvaluebool = value;
                    }

                    public DV(int vtype, String scope, String value) { // for
local variables
                           varvalue = 0; // default
                           vartype = vtype;
                           if (scope.equals("shared")) { shared = true; }
                           if (scope.equals("unshared")) { unshared = true; }
                           if (scope.equals("const")) { constant = true; }
                           if (value.equals("ID")) {varvalue =
DV.getLong(Runf.moteId());} }
```

Figure A1 continued

```
                    public DV(int vtype, String scope, structClass value) { //
for local variables
                        varvalue = 0; // default
                           vartype = vtype;
                           if (scope.equals("shared")) { shared = true; }
                           if (scope.equals("unshared")) { unshared = true; }
                           if (scope.equals("const")) { constant = true; }
                           if (vtype==4) {
                                 DVStructVal = new structClass();
                                 DVStructVal = value;}
                    }

                    public DV(String sourceId1, DV sourceVar1, int
targetRange1) { // for binding variables
                    varvalue = 0; // default
                    sourceId = sourceId1;
                    sourceVar = sourceVar1;
                    targetRange = targetRange1;
                    //bound = false;
                    //mote_id = -99;
                    }

                    public void push(DV x) {
                           valD.push(x);
                    }

                    public DV pop() {
                           DV vald = new DV(0, "unshared", 0);
                           if (!valD.isEmpty()) {vald = valD.pop(); return
vald;}
                           else {System.out.println("Stack is Empty!");
System.exit(0); return vald;}
                    }

                    public static boolean getBoolean(DV a) {
                           if (a.varvalue == 0) {return false;}
                           if (a.varvalue == 1) {return true; }
                           else {System.out.println("Not a Boolean!");
System.exit(0); return false;}
                    }

                    public static long getLong(DV a) {
                           return a.varvalue;
                    }

                    public static long type (int vtype, long value) {
                           if (vtype == 0) {return (long)value & 0xff; }
                           if (vtype == 1) {return (long)value & 0xffff; }
                           if (vtype == 2) {return (long)value & 0xffffffff; }
                           if (vtype == 3) {return (long)value & 0xff; }
                           return (long)value;
                    }

                    public static DV Constant(long a) {
                           DV val = new DV(0, "unshared", a);
                           return val;
                    }
                    public static DV ConstBool(boolean a) {
```

Figure A1 continued

```
       DV val = new DV(3, "unshared", 0);
       if (a == false) {val.varvalue = 0; }
       if (a == true) {val.varvalue = 1; }
       return val;
}

public static DV add(DV a, DV b) {
  long val1 = a.varvalue;
  long val2 = b.varvalue;
  long value;
  int size;
  String scope;

  if (a.vartype > b.vartype) {size = a.vartype;}
  else {size = b.vartype;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 + val2;
  result.varvalue = type(size, value);
  return result;
}

public static DV minus(DV a, DV b) {
  long val1 = a.varvalue;
  long val2 = b.varvalue;
  int size;
  long value;
  String scope;

  if (a.vartype > b.vartype) {size = a.vartype;}
  else {size = b.vartype;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 - val2;
  result.varvalue = type(size, value);
  return result;
}

public static DV mult(DV a, DV b) {
  long val1 = a.varvalue;
  long val2 = b.varvalue;
  int size;
  long value;
  String scope;

  if (a.vartype > b.vartype) {size = a.vartype;}
  else {size = b.vartype;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 * val2;
  result.varvalue = type(size, value);
  return result;
}

public static DV div(DV a, DV b) {
  long val1 = a.varvalue;
  long val2 = b.varvalue;
  int size;
```

Figure A1 continued

```
     long value;
     String scope;

     if (a.vartype > b.vartype) {size = a.vartype;}
     else {size = b.vartype;}
     scope = "unshared";
     DV result = new DV(size, scope, 0);
     value = val1 / val2;
     result.varvalue = type(size, value);
     return result;
   }

   public static DV mod(DV a, DV b) {
            long val1 = a.varvalue;
            long val2 = b.varvalue;
            int size;
            long value;
            String scope;

            if (a.vartype > b.vartype) {size = a.vartype;}
            else {size = b.vartype;}
            scope = "unshared";
            DV result = new DV(size, scope, 0);
            value = val1 % val2;

            result.varvalue = type(size, value);
            return result;
          }

   public static DV Eq(DV a, DV b) {
            long val1 = a.varvalue;
            long val2 = b.varvalue;
            DV result = new DV(3, "unshared", 0);

            if (val1 == val2) {
                   result.varvalue = 1;}
            return result;
          }

   public static DV Gt(DV a, DV b) {
     long val1 = a.varvalue;
     long val2 = b.varvalue;
     DV result = new DV(3, "unshared", 0);

     if (val1 > val2) {
     result.varvalue = 1; }

     return result;
   }

   public static DV Gte(DV a, DV b) {
     long val1 = a.varvalue;
     long val2 = b.varvalue;
DV result = new DV(3, "unshared", 0);

     if (val1 >= val2) {
     result.varvalue = 1; }
     return result;}
```

Figure A1 continued

```
                    public static DV Lt(DV a, DV b) {
                      long val1 = a.varvalue;
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 < val2) {
                      result.varvalue = 1; }
                      return result;
                    }

                    public static DV Lte(DV a, DV b) {
                      long val1 = a.varvalue;
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 <= val2) {
                      result.varvalue = 1; }
                      return result;
                    }

                    /*************************(int a, DV
b)**********************/

                    public static DV add(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
                      int size;
                      long value;
                      String scope;

                      if (4 > b.vartype) {size = 4;}
                      else {size = b.vartype;}
                      scope = "unshared";
                      DV result = new DV(size, scope, 0);
                      value = val1 + val2;
                      result.varvalue = type(size, value);
                      return result;
                    }

                    public static DV minus(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
                      int size;
                      long value;
                      String scope;

                      if (4 > b.vartype) {size = 4;}
                      else {size = b.vartype;}
                      scope = "unshared";
                      DV result = new DV(size, scope, 0);
                      value = val1 - val2;
                      result.varvalue = type(size, value);
                      return result;
                    }

                    public static DV mult(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
```

Figure A1 continued

```
  int size;
  long value;
  String scope;

  if (4 > b.vartype) {size = 4;}
  else {size = b.vartype;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 * val2;
  result.varvalue = type(size, value);
  return result;
}

public static DV div(int a, DV b) {
  long val1 = a;
  long val2 = b.varvalue;
  int size;
  long value;
  String scope;

  if (4 > b.vartype) {size = 4;}
  else {size = b.vartype;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 / val2;
  result.varvalue = type(size, value);
  return result;
}

public static DV mod(int a, DV b) {
         long val1 = a;
         long val2 = b.varvalue;
         int size;
         long value;
         String scope;

         if (4 > b.vartype) {size = 4;}
         else {size = b.vartype;}
         scope = "unshared";
         DV result = new DV(size, scope, 0);
         value = val1 % val2;

         result.varvalue = type(size, value);
         return result;
       }

public static DV Eq(int a, DV b) {
         long val1 = a;
         long val2 = b.varvalue;
         DV result = new DV(3, "unshared", 0);

         if (val1 == val2) {
                result.varvalue = 1;}
         return result;
       }

public static DV Gt(int a, DV b) {
  long val1 = a;
```

Figure A1 continued

```
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 > val2) {
                      result.varvalue = 1; }

                      return result;
                    }

                    public static DV Gte(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
                 DV result = new DV(3, "unshared", 0);

                      if (val1 >= val2) {
                      result.varvalue = 1; }
                      return result;
                    }

                    public static DV Lt(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 < val2) {
                      result.varvalue = 1; }
                      return result;
                    }

                    public static DV Lte(int a, DV b) {
                      long val1 = a;
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 <= val2) {
                      result.varvalue = 1; }
                      return result;
                    }



       /************************************************************/

                    /*************************(DV a, int
b)**********************/


                    public static DV add(DV a, int b) {
                      long val1 = a.varvalue;
                      long value;
                      int size;
                      String scope;

                      if (a.vartype > 4) {size = a.vartype;}
                      else {size = 4;}
                      scope = "unshared";
                      DV result = new DV(size, scope, 0);
```

Figure A1 continued

```
  value = val1 + b;
  result.varvalue = type(size, value);
  return result;
}
public static DV minus(DV a, int b) {
  long val1 = a.varvalue;
  long value;
  int size;
  String scope;

  if (a.vartype > 4) {size = a.vartype;}
  else {size = 4;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 - b;
  result.varvalue = type(size, value);
  return result;
}

public static DV mult(DV a, int b) {
  long val1 = a.varvalue;
  long value;
  int size;
  String scope;

  if (a.vartype > 4) {size = a.vartype;}
  else {size = 4;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 * b;
  result.varvalue = type(size, value);
  return result;
}

public static DV div(DV a, int b) {
  long val1 = a.varvalue;
  long value;
  int size;
  String scope;

  if (a.vartype > 4) {size = a.vartype;}
  else {size = 4;}
  scope = "unshared";
  DV result = new DV(size, scope, 0);
  value = val1 / b;
  result.varvalue = type(size, value);
  return result;
}

public static DV mod(DV a, int b) {
  long val1 = a.varvalue;
  long val2 = b;
  long value;
  int size;
  String scope;

  if (a.vartype > 4) {size = a.vartype;}
  else {size = 4;}
```

Figure A1 continued

```
     scope = "unshared";
     DV result = new DV(size, scope, 0);
     value = val1 % val2;

     result.varvalue = type(size, value);
     return result;
          }

   public static DV Eq(DV a, int b) {
      long val1 = a.varvalue;
      long val2 = b;
      DV result = new DV(3, "unshared", 0);

      if (val1 == val2) {
            result.varvalue = 1;}
      return result;
          }

   public static DV Gt(DV a, int b) {
     long val1 = a.varvalue;
     long val2 = b;
     DV result = new DV(3, "unshared", 0);

     if (val1 > val2) {
     result.varvalue = 1; }

     return result;
   }

   public static DV Gte(DV a, int b) {
     long val1 = a.varvalue;
     long val2 = b;
DV result = new DV(3, "unshared", 0);

     if (val1 >= val2) {
     result.varvalue = 1; }
     return result;
   }

   public static DV Lt(DV a, int b) {
     long val1 = a.varvalue;
     long val2 = b;
     DV result = new DV(3, "unshared", 0);

     if (val1 < val2) {
     result.varvalue = 1; }
     return result;
   }

   public static DV Lte(DV a, int b) {
     long val1 = a.varvalue;
     long val2 = b;
     DV result = new DV(3, "unshared", 0);

     if (val1 <= val2) {
     result.varvalue = 1; }
     return result;
   }
```

Figure A1 continued

```
/************************************************************/

             public static DV add(int a, int b) {
               long val1 = a;
               long val2 = b;
               long value;
               int size =4;
               String scope;

               scope = "unshared";
               DV result = new DV(size, scope, 0);
               value = val1 + val2;
               result.varvalue = type(size, value);
               return result;
             }

             public static DV minus(int a, int b) {
               long val1 = a;
               long val2 = b;
               long value;
               int size =4;
               String scope;

               scope = "unshared";
               DV result = new DV(size, scope, 0);
               value = val1 - val2;
               result.varvalue = type(size, value);
               return result;
             }

             public static DV mult(int a, int b) {
               long val1 = a;
               long val2 = b;
               long value;
               int size =4;
               String scope;

               scope = "unshared";
               DV result = new DV(size, scope, 0);
               value = val1 * val2;
               result.varvalue = type(size, value);
               return result;
             }

             public static DV div(int a, int b) {
               long val1 = a;
               long val2 = b;
               long value;
               int size =4;
               String scope;

               scope = "unshared";
               DV result = new DV(size, scope, 0);
               value = val1 / val2;
               result.varvalue = type(size, value);
               return result;           }

             public static DV mod(int a, int b) {
```

Figure A1 continued

```
     long val1 = a;
     long val2 = b;
     long value;
     int size =4;
     String scope;

     scope = "unshared";
     DV result = new DV(size, scope, 0);
     value = val1 % val2;

     result.varvalue = type(size, value);
     return result;
          }

   public static DV Eq(int a, int b) {
      long val1 = a;
      long val2 = b;
      DV result = new DV(3, "unshared", 0);

      if (val1 == val2) {
            result.varvalue = 1;}
      return result;
          }

   public static DV Gt(int a, int b) {
     long val1 = a;
     long val2 = b;
     DV result = new DV(3, "unshared", 0);

     if (val1 > val2) {
     result.varvalue = 1; }

     return result;
   }

   public static DV Gte(int a, int b) {
     long val1 = a;
     long val2 = b;
DV result = new DV(3, "unshared", 0);

     if (val1 >= val2) {
     result.varvalue = 1; }
     return result;
   }

   public static DV Lt(int a, int b) {
     long val1 = a;
     long val2 = b;
     DV result = new DV(3, "unshared", 0);

     if (val1 < val2) {
     result.varvalue = 1; }
     return result;
   }

   public static DV Lte(int a, int b) {
     long val1 = a;
     long val2 = b;
```

Figure A1 continued

```
                      DV result = new DV(3, "unshared", 0);

                      if (val1 <= val2) {
                      result.varvalue = 1; }
                      return result;
                    }



       /*******************************************************/

                    public static DV And(DV a, DV b) {
                           DV result = new DV(3, "unshared", 0);
                           boolean val1 = false;
                           boolean val2 = false;

                           if (a.vartype == 3) {if (a.varvalue == 1) {val1 =
true; } else {val1 = false; } }
                           if (b.vartype == 3) {if (b.varvalue == 1) {val2 =
true; } else {val2 = false; } }

                             if (val1 & val2) {
                             result.varvalue = 1; }
                           return result;
                     }

                    public static DV Or(DV a, DV b) {
                           DV result = new DV(3, "unshared", 0);
                           boolean val1 = false;
                           boolean val2 = false;

                           if (a.vartype == 3) {if (a.varvalue == 1) {val1 =
true; } else {val1 = false; } }
                           if (b.vartype == 3) {if (b.varvalue == 1) {val2 =
true; } else {val2 = false; } }

                           if (val1 || val2) {
                                    result.varvalue = 1; }
                           return result;
                    }

                    public static DV Ne(DV a, DV b) {
                      long val1 = a.varvalue;
                      long val2 = b.varvalue;
                      DV result = new DV(3, "unshared", 0);

                      if (val1 != val2) {
                             result.varvalue = 1; }
                      return result;
                    }

                    public static DV Not(DV a) {
                             boolean val = false;
                             DV result = new DV(3, "unshared", 0);
                             if (a.vartype == 3) {if (a.varvalue == 1) {val =
true; } else {val = false; } }
                             if( !val) {
```

Figure A1 continued

```
                           result.varvalue = 1;
                      }
                      return result;
                    }
             public static void set(DV a, DV b) {
               int size;

               if (a.vartype > b.vartype) {size = a.vartype;}
               else {size = b.vartype;}
               a.varvalue = b.varvalue;
             }
        public static void set(DV a, int b) {
               a.varvalue = b;
             }
        public static void set(int a, int b) {
               a = b;
             }
        public static void set(long a, DV b) {
               a = b.varvalue;
             }

        public static void set(DV a, boolean b) {
               a.varvaluebool = b;
             }

}
       //Declare Main
       public static void main(String[] args)
         {
             Thread t0= new Thread(new Body0() );
             t0.start();
             Thread t1= new Thread(new Body1() );
             t1.start();


         }
}
```

Figure A1 continued

APPENDIX B

DESAL GRAMMAR (GRAMMAR.DEF FILE)

The figure B1 shows some of the grammar rules in the grammar.def file. We invented a small, abbreviated syntax to generate dparser grammar, which is a style of commented Python. Here, the abbreviated syntax has two basic forms, single rules and multiple rulesets. Each rule or ruleset starts with a string, in column 1, looking like #nn#, where 'nn' is a number. The meaning of this number is taken from the Java Class numbering given by Dalton and Hallstrom, in their original Java compiler. An example is the following rule:

```
#2#
  componentDec: "component" var_id subComponentListNull
      if type(term2) == types.ListType:
              term2=term2[0]
 node["var_id"] = term1
 node["subComponentListNull"] = term2
 node["name"] = "component"
```

Notice that the rule starts with #2#, which means this will generate a dparser pattern (the first line following the #2# is the pattern), which eventually will generate a node of type 2, when a DESAL program is compiled into an Abstract Syntax Tree. The remaining lines in the rule are Python statements executed after dparser matches to the pattern. Here, some conventions are:

1. numTerms is a Python local variable equal to the number of terms matched by the dparser pattern. This may be variable, because dparser rules can have optional matching expressions (so numTerms isn't always the same number).

2. term0, term1, term2, etc, refer to the terms matched by the dparser pattern. You can refer to these terms in string manipulation and comparison code, but sometimes

the terms are not Strings, but are lists (this depends on how your grammar is defined). Notice above, the assignment: term2=term2[0]. This assignment is based on the assumption that term2 is a list prior to the assignment, and the first item of the list replaces local variable term2 (of course, this assumption is valid because of the "if" test on term2's type!).

3. The "output" of the rule is always a Python dictionary called, locally, 'node'. Here, you can add particular key/value things to this dictionary. Notice that above, the key "var_id" is added to the node. Some keys are standard, and be careful about these:

1. node["CaseNo"] -- automatically assigned, this will be the node's number (for the example above, it is 2).

2. node["LineNo"] -- automatically assigned, this is the line number of the DESAL program source for the matching program fragment.

3. node["ColumnNo"] -- like LineNo, but for column number.

4. node["Name"] -- optional; used basically for debugging and pretty printing of the parse tree by some tools.

For some cases, one dparser pattern could possible generate different node numbers, depending on inputs. To allow this, we have rulesets. A ruleset starts, like a rule, with a #xxx#-string in column 1, but the 'xxx' here will be a comma-separated list of numbers; these are the possible node numbers for the ruleset. Example:

```
#3,4#
 subComponentListNull: subComponentList?
   if not term0:
       #4#
   else:
     if type(term0) == types.ListType: term0=term0[0]
     #3#
     node["subComponentList"] = term0
```

This example shows a ruleset for node types 3 and 4. Notice that we see Python code interwoven with #3# and #4#, which are indicators of the specific definitions for node types 3 and 4. In the example, node type 4 has no special dictionary key/value items added, whereas type 3 has one key/value item added. Our tool validates that the lines following a ruleset definition contain entries for all the rules that should be defined. Thus, following #3,4#, there has to be some line #3# and some line #4# (and, of course, no line #5# or other crazy numbers).

```
       #0#
          program: Dispatch
          node["Dispatch"] = term0
          node["name"] = "program"

       #1#
          Dispatch: componentDec
          node["componentDec"] = term0
          node["name"] = "Dispatch"

       #2#
          componentDec: "component" var_id subComponentListNull
          if type(term2) == types.ListType: term2=term2[0]
          node["var_id"] = term1
          node["subComponentListNull"] = term2
          node["name"] = "componentDec"

       #3,4#
          subComponentListNull: subComponentList?
          if not term0:
              #4#
              node["name"] = "subComponentListNull"
          else:
              if type(term0) == types.ListType: term0=term0[0]
              #3#
              node["subComponentList"] = term0
              node["name"] = "subComponent"

       #5,6#
          subComponentList: subComponentList subComponent | subComponent
          if numTerms == 2:
             if type(term0) == types.ListType:
               term0=term0[0]
             #5#
             node["subComponentList"] = term0
             node["subComponent"] = term1
             node["name"] = "subComponentList"
          elif numTerms == 1:
             #6#
             node["subComponent"] = term0
             node["name"] = "subComponent"

       #7,8#
          subComponent: DecList Body | Body
          if numTerms == 2:
              #7#
              node["DecList"] = term0
              node["Body"] = term1
              node["name"] = "DecList"
          else:
              #8#
              node["Body"] = term0
              node["name"] = "Body"

       #9,10,11,114,118,12,13,14,115,119#
          DecList:  DecList StructInit | DecList structDec | DecList stateDec |
DecList bindingDec | DecList funcDec | StructInit  | structDec | stateDec |
bindingDec | funcDec
```

Figure B1 Grammar.def file containing the DESAL grammar rules.

```
   if numTerms == 2:
     if type(term0) == types.ListType:
        term0=term0[0]
     if  term1["CaseNo"] == 94:
       #print "\n\n "
       #print "struct Dec in DecList is: ", term1
       #print "\n\n "
       #9#
       node["DecList"] = term0
       node["strucDec"] = term1
       node["name"] = "DecList"
     if  term1["CaseNo"] == 15:
       #10#
       node["DecList"] = term0
       node["stateDec"] = term1
       node["name"] = "DecList"
     if  term1["CaseNo"] == 16:
       #11#
       node["DecList"] = term0
       node["bindingDec"] = term1
       node["name"] = "DecList"
     if  term1["CaseNo"] == 112:
       #114#
       node["DecList"] = term0
       node["funcDec"] = term1
       node["name"] = "DecList"
     if  term1["CaseNo"] == 85:
         #118#
         node["DecList"] = term0
         node["structInit"] = term1
         node["name"] = "DecList"
   if numTerms == 1:
     if  term0["CaseNo"] == 94:
       #12#
       #print "\n\n "
       #print "struct Dec is: ", term0
       #print "\n\n "
       node["structDec"] = term0
       node["name"] = "strucDec"
     if  term0["CaseNo"] == 15:
       #13#
       node["stateDec"] = term0
       node["name"] = "stateDec"
     if  term0["CaseNo"] == 16:
       #14#
       node["bindingDec"] = term0
       node["name"] = "bindingDec"
     if  term0["CaseNo"] == 112:
       #115#
       node["funcDec"] = term0
       node["name"] = "funcDec"
     if  term0["CaseNo"] == 85:
       #119#
       node["structInit"] = term0
       node["name"] = "structInit"

#15#
   stateDec: varClass varDec
```

Figure B1 continued

```
          node["varClass"] = term0
          node["varDec"] = term1
          node["name"] = "stateDec"

       #16#
          bindingDec: "binding" VarType bindingVarList
          node["VarType"] = term1
          node["bindingVarList"] = term2
          node["name"] = "bindingDec"

       #17,18#
          bindingVarList: bindingVarList ',' bindingVar | bindingVar
          if numTerms == 3:
            if type(term0) == types.ListType:
              term0=term0[0]
              #17#
              node["bindingVarList"] = term0
              node["bindingVar"] = term2
              node["name"] = "bindingVarList"
          if numTerms == 1:
            #18#
            node["bindingVar"] = term0
            node["name"] = "bindingVar"

       #19#
          bindingVar: var_id bindingExp
          node["var_id"] = term0
          node["bindingExp"] = term1
          node["name"] = "bindingVar"

       #20#
          bindingExp: bindingType bindingScope '.' var_id '.' var_id
bindingLimitExp
          node["bindingType"] = term0
          node["bindingScope"] = term1
          node["var_id"] = term3
          node["var_id1"] = term5
          node["bindingLimitExp"] = term6
          node["name"] = "bindingExp"

       #21#
          bindingType: "<-"
          node["name"] = "<-"

       #22,23#
          bindingScope: '*' | var_int
          if  term0 == '*':
             #22#
             node["name"] = "*"
          else:
             #23#
             #node[''expr''] = term0[''name'']
             node["name"] = term0["name"]

       #24,25#
          bindingLimitExp: ('[' var_int ']')?
          if not  term0:
               #25#
```

Figure B1 continued

```
        node["name"] = "bindingLimitExpNull"
   else:
        #24#
        node["expr"] = term0[1]["name"]
        node["name"] = "bindingLimitExp"

#26#
   Body: "every" expr TimeUnit "after" expr TimeUnit GuardList
   node["expr"] = term1
   node["TimeUnit"] = term2
   node["expr1"] = term4
   node["TimeUnit1"] = term5
   node["GuardList"] = term6
   node["name"] = "Body"

#27,28#
   GuardList: GuardList '[' ']' Guard | Guard
   if numTerms == 4:
       if type(term0) == types.ListType:
          term0=term0[0]
       #27#
       node["GuardList"] = term0
       node["Guard"] = term3
       node["name"] = "GuardList"
   else:
       #28#
       node["Guard"] = term0
       node["name"] = "Guard"

#29#
   Guard: expr ':' stmntListNull
   if type(term2) == types.ListType:
      term2=term0[0]
   node["expr"] = term0
   node["stmntListNull"] = term2
   node["name"] = "Guard"

#30,31#
   stmntListNull : stmntList?
   if not  term0:
        #31#
        node["name"] = "stmntListNull"
   else:
        if type(term0) == types.ListType:
          term0=term0[0]
        #30#
        node["stmntList"] = term0
        node["name"] = "stmntList"

#32,33,34#
   stmntList : stmntList stmnt | stmnt | "error"
   if numTerms == 2:
       #32#
       node["stmntList"] = term0
       node["stmnt"] = term1
       node["name"] = "stmntList"
   if numTerms == 1:
        if t != "error":
```

Figure B1 continued

```
                #33#
                node["stmnt"] = term0
                node["name"] = "stmnt"
               else:
                #34#
                node["name"] = "error"


       #35,36,37,38,117#
          stmnt : AssignS | ForEachS | IfS | FuncCallstmnt | funcCall
          if  term0["CaseNo"] == 40:
              #35#
              node["AssignS"] = term0
              node["name"] = "AssignS"
          if  term0["CaseNo"] == 41:
              #36#
              node["ForEachS"] = term0
              node["name"] = "ForEachS"
          if  term0["CaseNo"] >= 42 and  term0["CaseNo"]<=45:
              #37#
              node["IfS"] = term0
              node["name"] = "IfS"
          if  term0["CaseNo"] == 39:
              #38#
              node["FuncCallstmnt"] = term0
              node["name"] = "FuncCallstmnt"
          if  term0["CaseNo"] == 113:
              #117#
              node["FuncCall"] = term0
              node["name"] = "FuncCall"

       #39#
          FuncCallstmnt : FuncCall
          node["FuncCall"] = term0
          node["name"] = "FuncCallstmnt"

       #42,43,44,45#
          IfS :  "if"  expr  '{' stmntList '}' | "if"  expr  '{' stmntList '}'
Else | "if"  expr  '{' stmntList '}' ElseIfList | "if"  expr  '{' stmntList '}'
ElseIfList Else
          if numTerms == 5:
               #42#
               node["if"] = term0
               node["expr"] = term1
               node["stmntList"] = term3
               node["name"] = "If"
          if numTerms == 6:
                  if term5["CaseNo"] == 49:
                      #43#
                      node["if"] = term0
                      node["expr"] = term1
                      node["stmntList"] = term3
                      node["Else"] = term5
                      node["name"] = "If"
                  elif (term5["CaseNo"] == 46) or(term5["CaseNo"] == 47):
                      #44#
                      node["if"] = term0
                      node["expr"] = term1
```

Figure B1 continued

```
                      node["stmntList"] = term3
                      node["ElseIfList"] = term5
                      node["name"] = "If"
          if numTerms == 7:
               #45#
               node["if"] = term0
               node["expr"] = term1
               node["stmntList"] = term3
               node["ElseIfList"] = term5
               node["Else"] = term6
               node["name"] = "If"

       #52,53,54,55,56,57,58,59,60,61,62,63,115,64,65,66,67,68,69,70,71,72,73,74
,75,76,77#
          expr  : "ID" | SrcExpr | AgeExpr | BoundExpr | compoundvarName |
CastExpr | FuncCall | funcCall | BoolLit | var_int | var_id var_init | expr
'&&' expr $binary_left 2 | expr '||' expr  $binary_left 1 |'!' expr
$unary_right 7 | expr '<' expr $binary_left 4  | expr '>' expr $binary_left 4 |
expr '<=' expr $binary_left 4  | expr '>=' expr $binary_left 4 | expr '==' expr
$binary_left 3 | expr '+' expr $binary_left 5 | expr '-' expr $binary_left 5 |
expr '*' expr  $binary_left 6 | expr '/' expr  $binary_left 6 | expr '%'  expr
$binary_left 6 | '('  expr ')' $left 8 | AssignS | "error"
          if numTerms == 1:
               if  term0 == "ID":
                       #52#
                       node["expr_res"] = term0
                       node["name"] = "ID"
               elif  term0["CaseNo"] == 78:
                       #53#
                       node["expr_res"] = term0
                       node["name"] = "expr_res"
               elif  term0["CaseNo"] == 79:
                       #54#
                       node["expr_res"] = term0
                       node["name"] = "expr_res"
               elif  term0["CaseNo"] == 80:
                       #55#
                       node["expr_res"] = term0
                       node["name"] = "expr_res"
               elif  term0["CaseNo"] == 50 or  term0["CaseNo"] == 51 :
                       #56#
                       node["expr_res"] = term0
                       node["name"] = "expr_res"
               elif  term0["CaseNo"] == 82:
                       #57#
                       node["expr_res"] = term0
                       node["name"] = "cast"
               elif  term0["CaseNo"] == 81:
                       #58#
                       node["expr_res"] = term0
                       node["name"] = term0
               elif  term0["CaseNo"] == 88 or  term0["CaseNo"] == 89 :
                       #59#
                       node["expr_res"] = term0
                       #node["name"] = "Bool"
               elif  term0["CaseNo"] == 85:
                       #75#
                       node["expr_res"] = term0
```

Figure B1 continued

```
             #node["name"] = "exprList"
     elif  term0["CaseNo"] == 40:
             #76#
             node["expr_res"] = term0
             node["name"] = "="
     elif  term0 == "error":
             #77#
             node["name"] = "error"
     elif term0["CaseNo"] == 445:
             #60#
             node["expr_res"] = term0["var_int"]
             node["name"] = term0["var_int"]
if numTerms == 3:
     if  term1 == "&&":
             #61#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = "&&"
     elif  term1 == "||":
             #62#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = "||"
if (numTerms == 2) and ( term0 == "!"):
     #63#
     node["expr"] = term1
     node["name"] = "!"
if (numTerms == 2) and ( type(term0) == dict):
     #115#
     node["var_id"] = term0
     node["var_init"] = term1
if numTerms == 3:
     if  term1 == "<":
             #64#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = "<"
     elif  term1 == ">":
             #65#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = ">"
     elif  term1 == "<=":
             #66#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = "<="
     elif  term1 == ">=":
             #67#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = ">="
     elif  term1 == "==":
             #68#
             node["expr"] = term0
             node["expr1"] = term2
             node["name"] = "=="
     elif  term1 == "+":
```

Figure B1 continued

```
                #69#
                node["expr"] = term0
                node["expr1"] = term2
                node["name"] = "+"
        elif  term1 == "-":
                #70#
                node["expr"] = term0
                node["expr1"] = term2
                node["name"] = "-"
        elif  term1 == "*":
                #71#
                node["expr"] = term0
                node["expr1"] = term2
                node["name"] = "*"
        elif  term1 == "/":
                #72#
                node["expr"] = term0
                node["expr1"] = term2
                node["name"] = "/"
        elif  term1 == "%":
                #73#
                node["expr"] = term0
                node["expr1"] = term2
                node["name"] = "%"
   if numTerms == 3 and  term0 == "(":
                #74#
                node["lp"] = term0
                node["expr"] = term1
                node["name"] = "( )"

#78#
   SrcExpr : "src" '(' var ')'
   node["var"] = term2
   node["name"] = term2["var_id"]["var_id"]


#81#
   FuncCall :  "$" var_id ParmList
   node["var_id"] = term1
   node["ParmList"] = term2
   node["name"] = "Function Call"

#117,118#
   structVarList: structVarList ',' structVar | structVar
   #print " structVar is ", term0
   if numTerms == 3:
        if type(term0) == types.ListType:
           term0=term0[0]
        #117#
        node["structVarList"] = term0
        node["structVar"] = term2
        node["name"] = "structVarList"
   else:
        #118#
        node["structVar"] = term0
        node["name"] = "structVar"
#111#
   structType: var_id
```

Figure B1 continued

```
   node["structType"] = term0
#444#
   var_id : "[a-zA-Z_][a-zA-Z0-9_]*"
   node["var_id"] = term0
   node["name"] = term0

#446#
   itr_id : "[a-zA-Z_][a-zA-Z0-9_]*"
   node["itr_id"] = term0
   node["name"] = term0

#445#
   var_int : "0" | "[1-9][0-9]*"
   node["var_int"] = term0
   node["name"] = term0
```

Figure B1 continued

BIBLIOGRAPHY

1 Maurer , S.S.: A survey of embedded systems programming languages, Potentials, IEEE, Publication Date: Apr/May 2002, Volume: 21, Issue: 2, page(s): 30-34, ISSN: 0278-6648

2 Thomas, Philippe, Programming Embedded Systems: Seminar, WS 2006, Institute of Computer Science, University of Innsbruck

3 Bingham, Jeff and Lee Magnusson, H8/3664 based inertial rolling robot: Circuit Cellar feature article, Issue 200, March 2007

4 “Embedded Systems, from Wikipedia, the free encyclopedia”. http://en.wikipedia.org/wiki/Embedded_system (accessed Nov.30, 2009)

5 Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister: “System Architecture Directions for Networked Sensors,” ASPLOS, 2000

6 Gay, David, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler: The nesC Language: A Holistic Approach to Networked Embedded Systems, In Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

7 Yao, Y., and J. Gehrke: "The Cougar Approach to In-Network Query Processing in Sensor Networks," SIGMOD, 2002

8 Levis, Philip, and David E. Culler: Maté: a tiny virtual machine for sensor networks. ASPLOS 2002: 85-95

9 Liu, Jie, Maurice Chu, Juan Liu, James Reich, and Feng Zhao: “State-Centric Programming for Sensor-Actuator Network Systems.” in IEEE Pervasive Computing, October, 2003, pp.50-62.

10 Jonathan W. Hui and David Culler: The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. The 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04), November 3-5, 2004.

11 Klavins, Eric and Richard M. Murray: "Distributed Algorithms for Cooperative Control," IEEE Pervasive Computing, vol. 03, no. 1, pp. 56-65, Jan-Mar, 2004.

12 Zoumboulakis, M., Roussos, G., and Poulovassilis: A. Active rules for sensor databases. In Proceeedings of the 1st international Workshop on Data Management For Sensor Networks: in Conjunction with VLDB 2004 (Toronto, Canada, August 30 - 30, 2004). DMSN '04, vol. 72. ACM Press, New York, NY, 98-103.

13 Abdelzaher, Tarek F., Brian M. Blum, Qing Cao, Y. Chen, D. Evans, J. George, S. George, Lin Gu, Tian He, Sudha Krishnamurthy, Liqian Luo, Sang Hyuk Son, Jack Stankovic, Radu Stoleru, and Anthony D. Wood: EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. ICDCS 2004: 582-589

14 Jaein Jeong; and D. Culler: Incremental Network Programming for Wireless Sensors, IEEE SECON 2004, October 2004

15 Welsh Matt and Geoff Mainland: Programming Sensor Networks Using Abstract Regions. NSDI 2004: 29-42

16 Greenstein, B., E. Kohler, and D. Estrin: A sensor network application construction kit (SNACK). In Proceedings of the 2nd international Conference on Embedded Networked Sensor Systems (Baltimore, MD, USA, November 03 - 05, 2004). SenSys '04. ACM Press, New York, NY, 69-80.

17 Madden, S. R., M. J. Franklin, J. M.Hellerstein, and W. Hong: TinyDB: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30, 1(Mar.2005), 122-173.

18 Newton, R., Arvind, and M. Welsh: Building up to macroprogramming: an intermediate language for sensor networks. In Proceedings of the 4th international Symposium on information Processing in Sensor Networks (Los Angeles, California, April 24 - 27, 2005). Information Processing In Sensor Networks. IEEE Press, Piscataway, NJ, 6.

19 Fok, Chien-Liang, Gruia-Catalin Roman and Chenyang Lu: Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. ICDCS 2005: 653-662

20 Gummadi, Ramakrishna, Omprakash Gnawali and Ramesh Govindan: Macro-programming Wireless Sensor Networks Using Kairos. DCOSS 2005: 126-140

21 Han, Chih-Chieh, Ram Kumar, Roy Shea, Eddie Kohler and Mani B. Srivastava: A dynamic operating system for sensor nodes. MobiSys 2005: 163-176

22 Eswaran, Anand, Anthony Rowe, and Raj Rajkumar: Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. RTSS 2005: 256-265

23 Han, Chih-Chieh, Ram Kumar, Roy Shea, and Mani Srivastava: Sensor Network Software Update Management: A Survey in International Journal of Network Management, vol:15 , no:1099-1190 , pp:283-294 , 26 pages , John Wiley & Sons, Inc. , New York, NY, USA , July 2005. NESL Technical Report #: TR-UCLA-NESL-200503-09

24 Buonadonna, Phil, Joseph Hellerstein, Wei Hong, David Gay, Samuel Madden: TASK: Sensor Network in a Box. In Proceedings of European Workshop on Sensor Networks, 2005.

25 Liu, Jie, Elaine Cheong, and Feng Zhao: Semantics-Based Optimization Across Uncoordinated Tasks in Networked Embedded Systems. 5th ACM Conference on Embedded Software (EMSOFT 2005), EMSOFT '05, September 2005.

26 Whitehouse, K., F. Zhao, and J. Liu: Semantic Streams: a framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, 1 Microsoft Way, Redmond, WA 98052, April 2005.

27 Kwon, YoungMin, Sameer Sundresh, Kirill Mechitov, Gul Agha: "ActorNet: An Actor Platform for Wireless Sensor Networks," Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2006.

28 Gnawali, Omprakash, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan and Eddie Kohler: The TENET Architecture for Tiered Sensor Networks, In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (Sensys), November 2006.

29 Terfloth, K, G. Wittenburg, and J. Schiller.: FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks, First IEEE International Conference on Communication System Software and Middleware (COMSWARE 2006), New Delhi, India, January 2006

30 Gu, L. and J. A. Stankovic: t-kernel: Providing Reliable OS Support for Wireless Sensor Networks. In Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys'06), Nov, 2006.

31 Terfloth, K., G. Wittenburg, and J. Schiller: Rule-oriented Programming for Wireless Sensor Networks,International Conference on Distributed Computing in Sensor Networks (DCOSS) EAWMS Workshop, San Francisco, USA, June 2006

32 Razavi, Reza, Kirill Mechitov, Sameer Sundresh, Gul Agha and Jean-Francois Perrot: "Ambiance: Adaptive Object Model-based Platform for Macroprogramming Sensor Networks," ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2006.

33 McCartney, William P. and Nigamanth Sridhar: Abstractions for Safe Concurrent Programming in Networked Embedded Ssytems. Proceedings of SenSys '06: 4th ACM Conference on Embedded Networked Sensor Systems, November 1--3 2006. Pages 167--180.

34 Sen, Shondip and Rachel Cardell-Oliver: A Rule-Based Language for Programming Wireless Sensor Actuator Networks using Frequency and Communication. In proceedings of the third IEEE Workshop on Embedded Networked Sensors, Cambridge, MA. May 2006

35 Intagonwiwat, R. Gupta and A. Vahdat: "Declarative Resource Naming for Macroprogramming Wireless Networks of Embedded Systems", C. To appear in International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS), Venice, Italy, July 15 2006.

36 Kwon, YoungMin and Gul Agha: "Scalable Modeling and Performance Evaluation of Wireless Sensor Networks," to appear in Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE, 2006.

37 Woo, A., S. Seth, T. Olson, J. Liu, and F. Zhao: A spreadsheet approach to programming and managing sensor networks. In Proceedings of the Fifth international Conference on information Processing in Sensor Networks (Nashville, Tennessee, USA, April 19 - 21, 2006). IPSN '06. ACM Press, New York, NY, 424-431.

38 Razavi, Reza, Kirill Mechitov, Gul Agha and Jean-Francois Perrot: "Dynamic Macroprogramming of Wireless Sensor Networks with Mobile Agents," 2nd Workshop on Artificial Intelligence Techniques for Ambient Intelligence (AITAmI), 2007.

39 Hadim Salem and Nader Mohamed: "Middleware Challenges and Approaches for Wireless Sensor Networks," IEEE Distributed Systems Online, vol. 7, no. 3, 2006, art. no. 0603-o3001.

40 Arora, Anish, Mohamed Gouda, Jason Hallstrom, Ted Herman, Bill Leal, and Nigamanth Sridhar: A State-Based Language for Sensor-Actuator Networks, To appear, WWSNA 2007 (ACM/IEEE Workshop of IPSN 2007).

41 Prechelt, L: An Empirical Comparison of Seven Programming Languages. Computer 33, 10 (Oct. 2000), 23-29.

42 Dalton, Andy R., William P. McCartney, Kajari Ghosh-Dastidar, Jason O. Hallstrom, Nigamanth Sridhar, Ted Herman, William Leal, Anish Arora, and Mohamed Gouda: DESAL-a: An Implementation of the Dynamic Embedded Sensor-Actuator Language. In Proceedings of the International Conference on Computer Communications Networks (ICCCN 2008), US Virgin Islands, USA, August 2008.

43 GhoshDastidar, Kajari and Ted Herman: Separation of Circulating tokens, CoRR abs/0908.1797: (2009), SSS 2009.

44 Wang, L., and Y. Xiao: "A Survey of Energy-Efficient Scheduling Mechanisms in Sensor Networks", Mobile Network and Applications (MONET), 11(5), pp. 723-740, Oct. 2006.

45 Kumar, S., T. H. Lai, and J. Balogh: On k-coverage in a mostly sleeping sensor network. In Proceedings of the 10th Annual International Conference on M144–158, 2004.

46 Tian, D. and N. D. Georganas: A coverage-preserving node scheduling scheme for large wireless sensor networks. In Proceedings of the 1st ACM International Workshop on Wireless Applications (WSNA '02), pages 32–41, 2002.

47 Ye, F., G. Zhong, J. Cheng, S. Lu, and L. Zhang: Peas: A robust energy conserving protocol for long-lived sensor networks. In Proceedings of the 23rd International Conference (ICDCS '03), pages 28–37, 2003.

48 Cerpa and D. Estrin: Ascent: Adaptive self-configuring sensor networks topologies. In Proceedings of IEEE INFOCOM 2002, New York, NY, June 2002.

49 He, T., S. Krishnamurthy, L. Luo, T. Yan, L. Gu., R. Stoleru, G. Zhou, Q. Cao, P. , P. Stankovic,, T. Abdelzaher, J. Hui, and B. Krogh: VigilNet: An integrated sensor network system for energy-efficient surveillance. ACM Trans. Sen. Netw. 2, 1 (Feb. 2006), 1-38.

50 Yu, S. and Zhang, Y.: R-Sentry: Providing Continuous Sensor Services against Random Node Failures. In Proceedings of the 37th Annual IEEE/IFIP international Conference on Dependable Systems and Networks (June 25 - 28, 2007). DSN. IEEE Computer Society, Washington, DC, 235-244.

51 Dijkstra, E. W.: Self-Stabilizing Systems in Spite of Distributed Control, volume 17, pages 643–644. Communications of the ACM, 1974. 42

52 Herman, Ted, Chen Zhang: Best Paper: Stabilizing Clock Synchronization for Wireless Sensor Networks. SSS 2006: 335-349

53 Bapat, S., Leal, W., Kwon, T., Wei, P., and Arora, A. 2009. Chowkidar: Reliable and scalable health monitoring for wireless sensor network testbeds. *ACM Trans. Auton. Adapt. Syst.* 4, 1 (Jan. 2009), 1-32.

54 "TinyOS Tutorial". www.tinyos.net/tinyos-1.x/doc/tutorial/ (accessed Nov.30, 2009)

55 Nichols, M.H., A radio frequency identification system for monitoring coarse sediment particle displacement. Applied engineering in agriculture. 2004 Nov., v. 20, no. 6, p. 783-787.