ISSN No. 0976-5697

Volume 2, No. 4, July-August 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Implementing JAVA based Virtual Machine for Embedding Wireless Sensor Network Nodes

Jayesh N. Rathod*
Computer Science and Engineering Department
Laxmi Naraian Institute of Technology
Indore, India
jnrathod@aits.edu.in

Rashmi S. Agrawal
Computer Science and Engineering Department
Thakral Collage of Technology
Bhopal, India
Rashmi.agrawal@aits.edu.in

Abstract: On the design of a reliable programming model for wireless sensor networks (WSN), we must deal with various concerns, such as heterogeneousness of sensors, different sensing capabilities, dynamic updates and power consumption. The adhoc-networking characteristic of WSNs, its nonviable physical access, and the fact that WNS's are typically programmed in low-level paradigms, and the nonexistence of a robust semantic for existing languages are features that burden the task of programming sensor networks.

A more efficient approach to program WSN is using a high-level programming language combined with robust semantics. This combination is not provided by any existing programming languages. Consequently, it is not possible to prove the equivalence between the semantics of the language and its implementation. Therefore, a semantic gap is induced.

The CALLAS project proposes the creation of a calculus for a specific programming language and the corresponding virtual machine. Furthermore, it provides the semantic equivalence between the calculus and the virtual machine, thus the type-safety of the language. The main contribution of this thesis is the design and the implementation of a virtual machine for the Callas language, as derived from the base calculus

Keywords: Wireless, Sensor, Network, CALLAS, virtual machine, calculus

I. INTRODUCTION

Wireless Sensor Networks (WSNs) are collections of small and low-cost sensors, also called motes, as a result of their small size. These sensors can be deployed over wide areas and programmed to sense their environment. They can communicate data attributes over radio links using ad-hoc networking protocols [2]. It is expected that, in the future, thousands of low-cost motes, may be deployed over wide areas to provide long term monitoring of conditions and/or activity for days, months or even years.

Wireless Sensor Networks have been challenging the research community with an efficient programming model for them [3]. This programming model must deal with various concerns, such as: a heterogeneous mix of sensors, motes with different sensing capabilities, dynamic updates and power consumption [9]. WSNs have some very specific characteristics that are significantly different from other wireless networks, essentially:

A. The Design of a Sensor Network is Mostly Driven by the Target Application [5];

A. Sensor nodes are highly constrained in terms of CPU speed, memory availability and power consumption [5];

B. Large-scale sensor networks require self-configuration and automatic software updates without human intervention [5].

Moreover, the reliability in communication and lifetime of a WSN and its nodes is inferior compared to more conventional networks [16].

Wireless sensor networks have no rigid structure, and their granularity can vary from tens to thousands of nodes.

II. PROGRAMMING SENSOR NETWORKS

In this we have given a brief overview of a different variety of programming models for wireless sensor networks, focusing on their relative advantages and disadvantages.

The models can be broadly classified as high-level and low-level programming languages or tools [9].

A. Low-Level Programming:

The low-level programming of sensor networks can be done in distinct layers: directly using binary code, on top of a hardware abstract layer such as a virtual machine, and as part of a more generous middleware framework [15]. I have briefly described each approach, giving examples of programming languages/systems for each layer, and focusing on their advantages and disadvantages

B. Virtual Machines:

There are few virtual machine implementations for sensor networks. One of the better known in the literature is Mat, a communication-centric virtual machine [14].

The main focus of this virtual machine is the communication between sensors in the network [17]. Programming sensor networks with Mat´ can be done using the Tiny Script e language or using a higher level programming language called Mottle [10]. Programs are called capsules and they may be injected in the network, when needed, to achieve specific tasks [11]. These programs have the capability to move between sensors. There are also mechanisms that allow the installation of ad-hoc routing algorithms and data aggregation.

The Distributed Token Machine (DTM), used in the Regiment programming language, is another example of a virtual machine for sensor networks. DTM is a token driven virtual machine, in which, each token is a typed message with data/code that triggers a specific handler upon its reception [6]. All the execution and communication is based on event handlers. It has an associated intermediate language, called Token Machine Language (TML) that can be targeted by compilers for higher level systems.

C. High Level Programming:

At network level, the high-level programming can be divided in microprogramming or sensor-based programming [14]. The main idea of microprogramming systems is that applications for sensor networks should be developed as typical distributed applications without the need for the programmer to specify the role of each computing node individually [14]. The approach taken by these systems allows the programmer to focus on the application in a high-level fashion neglecting network architecture and communication details [7]. A compiler or a bundle of run-time libraries should take care of those details [13]. Applications in this kind of systems can be implemented using two types of behavior: global behavior and local behavior.

In global behavior, applications are implemented including all nodes in the network, without using any form of hierarchical partitioning of the network in subunits for the specification of the computation [6].

In systems using the local behavior approach, the network is partitioned in regions to specify a computation [17]. Abstract Regions, HOOD, Regiment, Kairos and SNACK are examples of systems and programming languages that are based on a local behavior approach. Regiment is a great example of this kind of language, as it uses network regions and data streams as the elemental programming abstractions. It is a functional language that does not permit input, output, or direct manipulation of a program state. The run-time environment is based on DTM virtual machine, previously described in this chapter [16].

On sensor-based programming systems, the responsibility to specify the role of each sensor (implementation, compilation and deployment) belongs to the programmer [10]. He must possess knowledge of the architecture of the sensor network and even perhaps the hardware of each sensor node. Smart Messages is based on this type of approach. Smart Message's allows messages between sensor nodes to carry code, data and execution state. It is based on Java and expands the Java APIs with some features, such as support for spatial programming.

Also in this line of work, project CALLAS tackles the problem of providing the WSN with a robust programming model in the sense that the languages obtained are type-safe and that the semantics of the virtual machine matches that of the formal model [9]. The type-safety property of the Callas language allows programs to be verified statically and a significant set of would-be run-time errors to be detected prematurely [8]. Higher level programming in CALLAS is achieved by encoding high-level constructs in the core programming model, thus preserving semantics.

III. THE PROGRAMMING MODEL

Callas is a calculus for programming sensor networks that provides primitives for sensor computation, communication, code mobility, and updates. It provides a type-safe framework for developing programming languages and run-time systems [14].

A network is represented as S or as 0, being the later the representation of an empty network [7]. A network is a concurrent composition of sensors devices, represented as $[C, R \triangleright M, T]^{I,O}_{p,t}$.

Each sensor device comprises the following elements [12]:

A. C - a call-stack for the running process;
B. R - a priority queue of runnable processes;
C. M - a table with the installed code modules;
D. T - a table of timers for function calls;
E. I - a queue of incoming messages from the network;
F. O - a queue of outgoing messages for the network;
G. p - the current position;
H. t - the current time.

The running process uses the C stack, while the runnable processes are located at R. The interface between low-level networking is done with I and O queues. These queues buffer the messages between sensor devices in the network. Messages are serialized or packaged function calls of the form l [5]. The devices are not only capable of measuring their position, p, but can also sense a few physical properties of the environment (e.g. temperature, humidity), by calling external routines. The code installed in each sensor is represented by M and it consists of a set of named functions. The later are represented by l = (x)P , where l is the name of the function, x are the parameters, and P the code for the function. T is a set of timed function calls to functions in M. Each timed call is a tuple formed by the call to be triggered, the timer period, the time after which the timer expires and, the time of the next call.

The values exchanged between sensor devices are represented by v, and comprise basic values b (primitive data-types from the sensors), and code M, representing byte-code modules [16].

A process P can: call a function (v .l (v)), call an external function (extern l (v)), install a module (M.install M′), send a message (send l(v)), receive a message (receive), program a timed call (timer l(v) every v expire v), or assign values to variables (the let construct). In the module installation, the process adds the set of functions in M′ to M. send l (v) takes a call, packages it, and places it at the outgoing-queue, to be sent over the network. On the other hand, a receive gets a packaged call from the incoming-queue, unpacks it, and places it in the run-queue. In a timed-call timer l (v) every v expire v, the call l (v) is executed periodically until the timer expires. A timed call that does not expire is written as timer l (v) every v forever. Finally, the let construct permits the processing of intermediate values in computations [15].

A. Semantics:

The operational semantics is defined with the help of a structural congruence as usually in process calculus the congruence rules are given in Figure 3.2 [17].

Standard rule is [C, R▷ M, T]Ip, , Ọ≡ [C, R ▷ M, T]Ip, ,O {0̣ }, which provides a conceptual membrane to the sensor. This membrane is an artifice to prevent sensors from receiving duplicate copies of a message during a broadcast [3].

All sensor's reductions are carried out without any interference. The installation of a module on a sensor is controlled by the rules R- INSTALL- INTERFACE and R-INSTALL – MODULE

S1 | S2 ≡ S2 | S1, S | 0 ≡ S, S1 | (S2 | S3) ≡ (S1 | S2) | S3 (S-monoid-Sensor)

$$[C, R \triangleright M, T]^{I,O}_{p,t} \equiv [C, R \triangleright M, T]^{I,O}_{p,t}\{\mathbf{0}\} \qquad (\textsc{S-Init-Send})$$

Figure 1: Structural congruence for sensors.

The first one is responsible for the installation of a module on the sensor's interface, and the second one is responsible for the installation of a module on an anonymous module.

B. *Language Syntax:*

We have presented some examples written in a concrete syntax derived from the calculus for the Callas language. The syntax does not use braces to delimit blocks. It is inspired in the Python programming language and its indentation style, which makes use of white space to delimit blocks. The grammar for the concrete syntax may be consulted in [18]. In the Callas Language, lines which end with (:) delimit the start of a block of code and the consequent lines with the same indentation constitute the body of the block.

The first example is the sampling program presented in the previous section, the second one is a simple Ping, and the last one computes a maximum value of a data attribute in a network [15].

```
#Sink
run:
module Sampling as sampling
def gather(self ,x ,y);
extern log(x ,y);
def receiver(self ,x ,y);
receive sensor. install(sampling);
send setup(period, interval);
timer receiver() every dt forever
```

Figure 2: Sampling program – *Sink*

```
#Sensor
run:
module Sampling as sampling
def setup(self ,x ,y):
fire self. sample() every x ,require y
def sample(self);
x=extern time()
y=extern data()
send gather(x ,y)
def receiver(self ,x ,y):
receive
sensor. install(sampling);
timer receiver() every dt forever
```

Figure 3: Sampling program – *Sensor*

IV. VIRTUAL MACHINE

The idea is to prove, in future work, the equivalence between the semantics of the calculus and that of the virtual machine, thus establishing the soundness of the virtual machine. The virtual machine we present serves as the specification for the run-time environment of the Callas programming language.

The programming language chosen to implement the virtual machine was Java. This choice was a result of the adoption of SunSPOT^R platform for the development. The SunSPOT^R devices run a compact version of a Java virtual machine, called Squawk.

A. *Format of Byte –Code:*

In the byte-code format, a program (p) has as its constituents: a magic-number, a version, a list of modules, and a list of types. Both magic-number and version are basic types [13].

A module (m) is a list of functions (f), and each function is composed by a string with its name, the number of local variables, the code with its instructions, and a set of constant symbols [13]. All the instructions are given by c, and each constant symbol is given by u, that can be one of the follow: BOOL k, INTEGER n, FLOAT e, STRING k l,

B. *Specification:*

The syntactic categories of the virtual machine are defined in Figure 4.2. A byte-array of Callas byte-code is represented by b of a set B. A word w of set (W) used in the virtual machine can be a boolean, an integer, a float, a string, or a module m of set M, which is a map String→B 7 representing a set of functions [11]. The virtual machine has both incoming, and outgoing queues of sets I and O for network purposes. Both queues manipulate frames of set F, which are sequences of words. A priority queue, of set R, is also present. This queue is a sequence of runnable processes of set H [12]. A runnable process is composed of a tuple with: an operand-stack of set S and a byte-array of set B. On the other hand, the running-process is executed in a call stack of set C that contains tuples, of set G, with: an operand-stack, byte-code, an environment frame, and a program counter [7]. Finally, timed-calls are represented as tuples, of set T , with an operand-stack and tree integers representing the period, expire time, and the time of the next call. Finally, a machine state is a tuple of set:
Int ×M× T × C × R × I × O ∪ {halt}

C. *Semantics:*

In this section we present the operational semantics of the Callas virtual machine. The semantics is based on the reduction semantics for the calculus presented in [5], and is parametric in the program p.

byte array	b ∈ B
word	w ∈ W = Bool ∪ Int ∪ Float ∪ String ∪M
frame	F = h ~W i
operand-stack	S = ~W
timer	T = S × Int × Int × Int
incoming-queue	I = ~F
outgoing-queue	O = ~F
runnable process	H = S × B
run-queue	R = ~H
running process	G = S × F × Int × B
call-stack	C =~G
module	m ∈ M= String 7→ B
machine state	Int ×M× T × C × R × I ×∪O {halt}

Figure 4: The syntactic categories of the virtual machine.

n-initialized k-extra frame	h<w>k≡<w1,...,wn, 01, ... , 0k>, k ≥ 0
frame	<w> ≡ <w>0
stacks of syntactic category	α α1 : · · · : αn |Q
queues of syntactic category	α α1 :: · · · :: αn | Q

Figure 5: Notation for the components of the virtual machine.

D. Implementation:

In this section we focus on the implementation of the virtual machine. The main objective was to use the virtual machine's operational semantics as a specification.

The data-structures we require map one-to-one with those of the formal semantics (Table 4.4). The implementation was done in the Java programming language, to run on the Squawk virtual machine installed in SunSPOT^R devices [9]. The source code of the virtual machine is provided with this thesis in Appendix A.

Tabal : 1 Data-Structures map

Abstraction	Implementation
b	byte[]
p	Program
m	Module
f	Function
$\mathcal{H}$	RunnableProcess
$\mathcal{G}$	RunningProcess
$\mathcal{T}$	TimedCall
$\mathcal{F}$	java.util.Vector
$\mathcal{I}$	com.sun.spot.util.Queue
$\mathcal{O}$	com.sun.spot.util.Queue
$\mathcal{R}$	com.sun.spot.util.Queue
$\mathcal{S}$	java.util.Stack
$\mathcal{C}$	java.util.Stack

The virtual machine's primary structure is the class Program, which keeps run-time information about the program's byte-code p.

```
Class  Program
private  int  magicNumber ;
private byte version ;
private  Vector modules ;
private   Vector types;
private  byte [ ] byteCode ;
Program(int magicNumber ,byte version,Vector modules ,Vector typesbyte [ ] byteCode)
        {
                . . .
        }
}
```

Class Program has the following elements: a magic number, a version, a vector that keeps the program's modules, and a vector that keeps the type information. The number of modules and types can be extracted from the Vector object.

As we saw, a Program has a list of modules and each element of this list has its own structure. Next, we present the definition of the Module class.

```
Class Module {
        protected byte open ;
        protected Vector functions;
        Module( )
        { . . .
        }
}
```

The class Module has only two elements: a byte that indicates whether a module has free variables, and a vector with the functions that make up the module [16]. To be more conservative in terms of memory we use a vector instead of a hash table, we consider that we do not have a large number of functions in a module. In future work, the use of a hash table will be preferable.

Class Function has the following attributes: the name, the number of locals, the bytecode, and the symbols. The function's symbols are constants that can be referred to in the byte-code and can be one of the following types: integer, float, boolean, string, or a module [9]. We extract the symbols associated with every function from its bytecode to make the implementation simpler. The field byteCode of the class Function should then be understood as composed of the machine instructions for the body of the function.

```
C lass Function
{
private  String name ;
private byte [ ] byteCode ;
private Vector symbols ;
private byte numberOfLocals;
Function( )
{ . . . }
}
```

These data-structures completely describe the program's byte code.

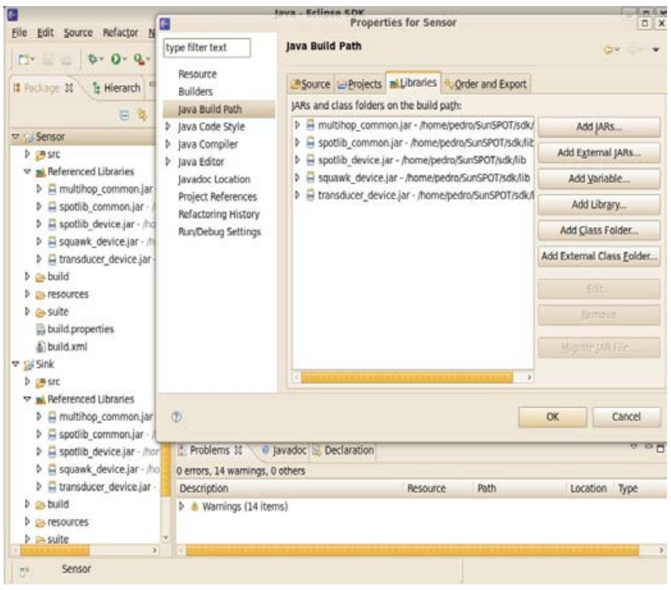
We now turn to the virtual machine data-structures. One of these structures is represented by class RunnableProcess, which represents a run-queue tuple H for a given function and environment frame.

E. Developing and Running Callas Applications:

For the development of the virtual machine for the Callas language and of Callas applications we have chosen the Eclipse platform [13]. The reasons behind this choice were the fact that it runs on multiple platforms and has an excellent support for Java based development.

Here, we have presented a small example session that builds Callas applications and runs it on the SunSPOT^R simulation tool Solarium. We will use the Ping program, and two sensors: Sensor and Sink. Both are Java projects, and have an embedded Callas virtual machine.

In Figure 4.5 we see the SunSPOT SDK libraries used by the virtual machine and the applications.



Java Build Path

JARs and class folders on the build path:

multihop_common.jar - /home/pedro/SunSPOT/sdk/
spotlib_common.jar - /home/pedro/SunSPOT/sdk/lib
spotlib_device.jar - /home/pedro/SunSPOT/sdk/lib
squawk_device.jar - /home/pedro/SunSPOT/sdk/lib
transducer_device.jar - /home/pedro/SunSPOT/sdk/l

Sensor
src
Referenced Libraries
multihop_common.jar
spotlib_common.jar
spotlib_device.jar
squawk_device.jar
transducer_device.jar
build
resources
suite
build.properties
build.xml
Sink
src
Referenced Libraries
multihop_common.jar
spotlib_common.jar
spotlib_device.jar
squawk_device.jar
transducer_device.jar
build
resources
suite

0 errors, 14 warnings, 0 others

Warnings (14 items)

Figure 6: Referenced libraries.

Each project has a build.xml file (same as the Demos application projects supplied with the SunSPOT SDK), and a MANIFEST.MF file with the following configuration:

```
MIDlet−Name: VirtualMachine
MIDlet−Version: 1. 0. 0
MIDlet−Vendor: Sun MicrosystemsInc
MIDlet −1: , , org.callas.vm.VirtualMachine
MicroEdition−Profile: IMP−1.0
MicroEdition−Configuration : CLDC−1.1
```

This indicates the entry point to the application.

An application is formed by the Java code of the virtual machine plus a byte-array that holds the Callas program byte-code. Upon initialization in the sensor, the virtual machine reads the byte-array and extracts this data into a run-time representation of the program.

The SPOT Emulator is able to run SunSPOT applications in desktops. Further, it has almost the same features as a real SunSPOT namely: a configurable sensor panel instead of a physical sensor board, configurable leds, and communication via radio [17].

To create a virtual SPOT we can use the interface, Figure 4.6.

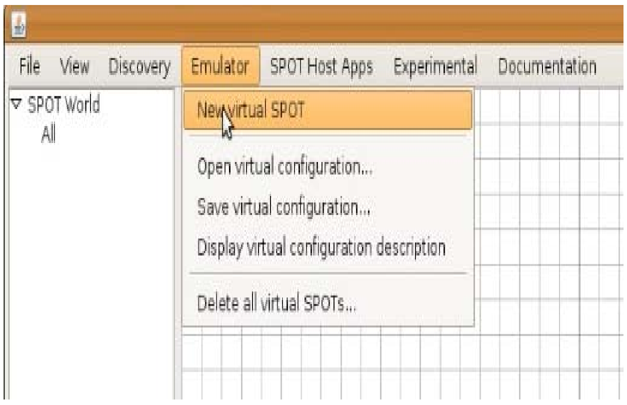
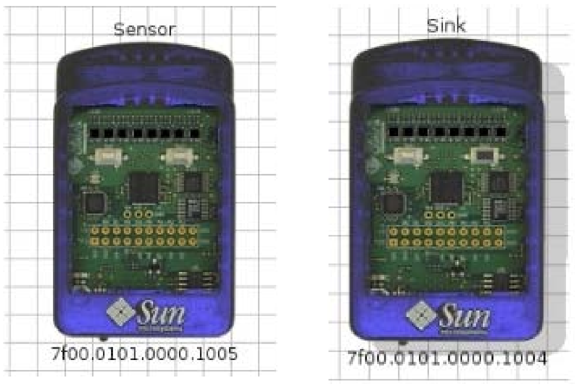

File
View
Discovery
Emulator
SPOT Host Apps
Experimental
Documentation
SPOT World
All
New virtual SPOT
Open virtual configuration...
Save virtual configuration...
Display virtual configuration description
Delete all virtual SPOTs...

Figure 7: Creating a virtual SunSPOT.

To proceed with the Ping example we hit the Deploy MIDlet bundle action (Figure 4.8), which allow us to deploy an application onto the virtual SPOT.We select the build.xml file corresponding to the project, or an existing project's jar file that we want to deploy to the virtual SPOT. Another action that we use, the Display application output, allows us to view the output from applications running on the virtual device. We can also use the Set Name command to label virtual SPOTs, in this case sensor or sink (Figure 4.9).



(a) Sensor (b) Sink

Figure 8: Set the name of the virtual SPOTs.

The application consists of two projects with the source of the Callas virtual machine and the byte-code to be executed.

Next, we set the name of each virtual SPOT: one as Sensor, the other as Sink. In Figure 4.9 we can view that the left virtual SPOT is the Sensor and the right is the Sink.

After this, we deploy both projects in their respective virtual SPOT (Figures 4.10 and 4.11) by selecting the appropriate build.xml files in each of the ping project/sensor and ping project/sink directories.

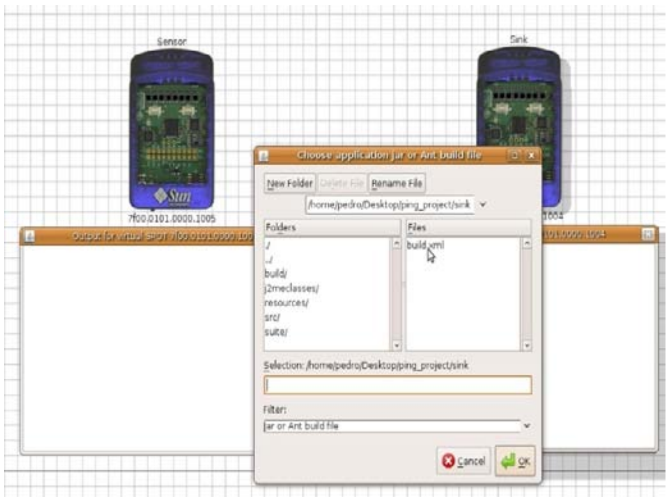


Figure 9: Deploy Sink project.

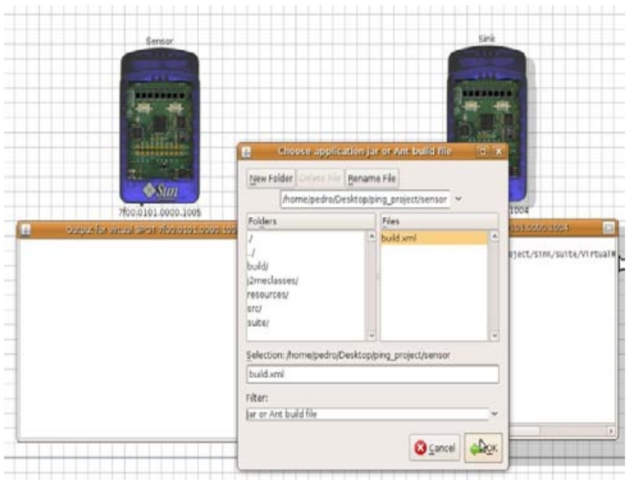


Figure 10: Deploy Sensor project.

The first one is responsible for the installation of a module on the sensor's interface, and the second one is responsible for the installation of a module on an anonymous module.

V. CONCLUSIONS

In this thesis we have presented the base calculus, presenting its reduction semantics, proved before being type-safe.

We have worked on specifying an operational semantics for a virtual machine for Callas based on the same calculus, maintaining as closest as possible from that of the calculus. Based on this operational semantics, we have developed an implementation of a virtual machine for the SunSPOT platform, which runs on top of a Squawk virtual machine.

We have written some programs in Callas language and created correspondent projects, and tested them in the SunSPOT devices, and in the Solarium tool. All of the programs written are type-safe.

For future work, we are intending to create high-level programming languages, add more abstractions to the semantics (e.g. regions), and prove the semantic equivalence between the calculus and the virtual machine, thus proving the type-safety of the language.

VI. ACKNOWLEDGMENT

Our first thanks are to the Almighty God, without whose blessings we wouldn't have been writing this "acknowledgments".

We then would like to express our heartfelt thanks to our guide, Prof. Arun Johar for giving us the guidance, encouragement, counsel throughout our re-search and painstakingly reading our reports. Without his invaluable advice and assistance it would not have been possible for us to complete this thesis.

Finally, we would like to thank all of them whose names are not mentioned here but have helped us in any way to accomplish the work.

VII. REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, _A survey on sensor networks,_ IEEE Communication Mag., vol. 40, no. 8,2002.

[3] I. F. Akyildiz and I. H. Kasimoglu, _Wireless sensor and actor networks: Research challenges,_ Ad Hoc Networks Journal, vol. 2, no. 4,2004.

[4] On World - Emerging Wireless Research, www.onworld.com.

[5] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, _The RUNES middleware for networked embedded systems and its applicationin a disaster management scenario,_ in Proc. of the 5th Int. Conf. on Pervasive Communications (PERCOM), 2007.

[6] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, _A reconfigurable component-based middleware for networked embedded systems,_ Int. Journal of Wireless InformationNetworks, vol. 14, no. 2, 2007.

[7] L. Mottola, G. P. Picco, and A. Amjad, Fine-grained software reconfiguration in wireless sensor networks,_ in Proc. of 5th European Conf. on Wireless Sensor Networks (EWSN), 2008.

[8] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, _TeenyLime: Transiently shared tuple space middleware for wireless sensor networks,_in Proc. of the 1st Int. Wkshp. on Middleware for Sensor Networks (MidSens), 2006.

[9] Programming wireless sensor networks with the TeenyLime middleware,_ in Proc. of the 8th ACM/USENIX Int. MiddlewareConf., 2007.

[10] L. Mottola and G. P. Picco, _Programming wireless sensor networks with Logical Neighborhoods,_ in Proc. of the 1st Int. Conf. on Integrated Internet Ad hoc and Sensor Networks (InterSense), 2006.

[11] Logical Neighborhoods: A programming abstraction for wireless sensor networks,_ in Proc. of the 2nd Int. Conf. on Distributed Computing on Sensor Systems (DCOSS), 2006.

[12] P. Ciciriello, L. Mottola, and G.P. Picco, _Building virtual sensors and actuator over Logical Neighborhoods,_ in Proc. of the 1st ACM Int. Wkshp. on Middleware for Sensor Networks (MidSens06 – colocated with ACM/ USENIX Middleware), 2006.

[13] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco,_Expressing sensor network interaction patterns using data-driven macroprogramming,_ in Proc. of the 3rd Int. Wkshp. on Sensor Networks and Systems for Pervasive Computing (PerSens – colocated with IEEE PERCOM), 2007.

[14] L. Mottola, A. Pathak, A. Bakshi, G. P. Picco, and V. K. Prasanna,_Enabling scope-based interactions in sensor network macroprogramming, _ in Proc. of the the 4th Int. Conf. on Mobile Ad-Hoc and Sensor Systems (MASS), 2007.

[15] M. Welsh and G. Mainland, _Programming sensor networks using abstract regions,_ in Proc. of 1st Symp. on Networked Systems Design and Implementation (NSDI), 2004.

[16] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, _Reliable and e_cient programming abstractions for wireless sensor networks,_in Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2007.

[17] R. Newton, G. Morrisett, and M. Welsh, _The Regiment Macroprogramming system,_ in Proc. of the 6th Int. Conf. on Information Processing in Sensor Networks (IPSN), 2007.

[18] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, _TinyDB: an acquisitional query processing system for sensor networks,_ ACM Trans. Database Syst., vol. 30, no. 1, 2005.