A Virtual Machine for Wireless Sensor Networks

Pedro Emanuel Rodrigues Gomes

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2009

A Virtual Machine for Wireless Sensor Networks

Pedro Emanuel Rodrigues Gomes

Dissertação Submetida à Faculdade de Ciências da Universidade do Porto
para obtenção do grau de Mestre em Engenharia de Redes e Sistemas Informáticos

Orientador: Professor Luís Lopes

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2009

To my family and girlfriend.

Acknowledgements

I would like to express my thanks to my dear girlfriend Juliana Fernandes, who have supported me all the time, and helped me to achieve all my purposes. I am very grateful to you.

I am very grateful for all of the help given by my friend and colleague João Almeida.

Furthermore, I appreciate all the support given by my colleagues Ricardo Fernandes, Diogo Ferreira, Hugo Conceição, Pedro Lopes, Ricardo Castro, Ricardo Batista, Pedro Ramalho, Paulo Pinto and Pedro Monteiro.

I would like to thank to my good friends Alexandra Maia, Vânia Rodrigues, Ricardo Costa, and Stephanie Mesquita for all the motivation and support given during this thesis.

I would like to thank my supervisor Prof. Luís Lopes for in 2008 offering me a research scholarship, and the possibility to work with him. Further, I would like to express my gratitude to my supervisor, for all of the support given in the work during this thesis.

To finalize, I formally would like to thank to the insightful talks and discussions with my colleague Tiago Cogumbreiro.

Resumo

No desenho de um modelo de programação fiável para redes de sensores temos de ter em conta várias preocupações como a heterogeneidade de sensores, diferentes capacidades de cada sensor, actualizações dinâmicas e consumo de energia. A complexidade na programação de sensores é enorme devido a uma série de factores, tais como, estas serem ad-hoc, estarem distribuídas por lugares de difícil acesso, serem tipicamente programadas usando paradigmas de programação de baixo nível, e o facto de não existir semânticas robustas para as actuais linguagens e as suas implementações.

Na programação de redes de sensores, uma abordagem mais eficiente advém do uso de uma linguagem de programação de alto nível combinada com uma semântica robusta. As actuais linguagens de programação não fornecem este combinação, existindo um salto a nível semântico entre a semântica da linguagem e a semântica da implementação, não havendo possibilidade de provar a sua equivalência semântica.

O projecto CALLAS propõe-se a criar um cálculo para uma linguagem de programação e a sua respectiva máquina virtual. Propõe-se também a eliminar o salto a nível semântico, provando a equivalência semântica entre o cálculo e a máquina virtual. A principal contribuição desta tese é o desenho e a implementação de uma máquina virtual para a linguagem Callas, sendo esta baseada no cálculo criado.

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, 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.

Contents

Acknowledgements **4**

Resumo **5**

Abstract **6**

List of Tables **9**

List of Figures **11**

1 Introduction **12**

1.1 Problem Statement 14

1.2 Goal 14

1.3 Outline 14

2 Programming Sensor Networks **16**

2.1 Low-level Programming 16

2.1.1 Binary Code 16

2.1.2 Virtual Machines 18

2.1.3 Middleware 21

2.2 High-level programming 22

3 The Programming Model **24**

3.1 Abstract Syntax 24

3.2 Semantics 27

3.3 Language Syntax 29

4 The Virtual Machine **34**

4.1 Format of byte-code 34

4.2 Specification 36

4.3 Semantics 36

4.4 Implementation 42

4.5 Developing and Running Callas applications 52

5 Conclusions **58**

A The Callas Virtual Machine. **59**

List of Tables

1.1 Applications of WSN 13

4.1 Data-structures map. 43

List of Figures

1.1 An example of a Mica mote. 12

2.1 An example of nesC programming. 17

2.2 An example of a service call in Contiki. 18

2.3 An example of a SunSPOT device. 19

2.4 IBM Mote Runner architecture. 20

2.5 IBM Mote Runner components. 21

3.1 The Callas abstract syntax. 26

3.2 Structural congruence for sensors. 28

3.3 Sampling program - Sink. 30

3.4 Sampling program - Sensor. 30

3.5 Ping program - Sink. 31

3.6 Ping program - Sensor. 31

3.7 Maximum value of data attribute example - Sensor. 33

4.1 The byte-code format. 35

4.2 The syntactic categories of the virtual machine. 37

4.3 Notation for the components of the virtual machine. 37

4.4 The initial state. 38

4.5 Referenced libraries. 53

4.6 Creating a virtual SunSPOT. 54

4.7 A new virtual SunSPOT. 54

4.8 Possible actions. 55

4.9 Set the name of the virtual SPOTs. 55

4.10 Deploy Sink project. 56

4.11 Deploy Sensor project. 56

4.12 Result of Ping program execution. 57

Chapter 1

Introduction

Wireless Sensor Networks (WSNs) are collections of small and low-cost sensors [1], also called *motes*, as a result of their small size (Figure 1.1). 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. 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.

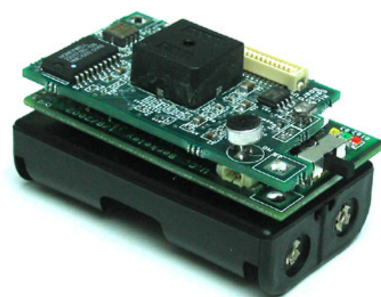


Figure 1.1: An example of a Mica mote.

Wireless Sensor Networks have been challenging the research community with an efficient programming model for them. 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. WSNs have some very specific characteristics that are significantly different from other wireless networks, essentially:

- the design of a sensor network is mostly driven by the target application;

- sensor nodes are highly constrained in terms of CPU speed, memory availability and power consumption;

- large-scale sensor networks require self-configuration and automatic software updates without human intervention.

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

Wireless sensor networks have no rigid structure, and their granularity can vary from tens to thousands of nodes [2].

The potential applications in WSN's are numerous. An overview of their wide range is displayed in Table 1.

Field	**Application**
environment	forest fire detection flood detection
medicine	health monitoring medical diagnostics
management	traffic control inventory control
physics	high-energy physics-particle detectors
military	reconnaissance of opposing forces and terrain nuclear, biological and chemical attack detection and reconnaissance
home	home automation smart environment
commercial	car theft monitoring interactive museums

Table 1.1: Applications of WSN

Considering this wide range of sensor network's applications, WSNs can and will be an integral part of our lives, providing the end user with intelligence and a better understanding of the environment.

1.1 Problem Statement

Reliable programming of WSNs is difficult, as a result of a set of issues, such as: ad-hoc networking, frequent impossibility of physical access to the sensors, low-level programming, and the absence of *type-safe* programming languages with provably correct operational semantics. While the first two factors are difficult to approach, a lot can be done to improve the robustness off current programming languages and their abstractions. The combination of a high-level programming language with a robust semantic framework, in our view, the way to proceed. Existing high-level programming languages do not have *type-safety* or provably correct semantics. There is a *semantic gap* between the languages and their implementation, which makes deployment and debugging of applications a complex task.

The main contribution of the CALLAS project, in which this work is included, is to produce a *type-safe*, semantically robust framework upon which to develop programming languages and a virtual machine for WSNs. Further, CALLAS contributes with the proof of semantic equivalence between the calculus and the virtual machine in future work, thus the *type-safety* of the language. Therefore, eliminating an appreciable set of errors in run-time, thus simplifying the deployment.

1.2 Goal

The goal of this thesis is to specify and implement a virtual machine for the Callas programming language, as defined from the base calculus [3]. The development of a virtual machine provides a portable run-time environment for Callas applications, and one that is scalable in the sense that *type-safety* and the robust semantics allow for changes in the size of the target sensor network.

1.3 Outline

In Chapter 2, we describe the state-of-the-art solutions for programming WSNs, including programming languages, run-time systems, and operating systems. Chapter 3 focuses on the programming model for WSNs, and its semantics. The main contribution of this thesis is presented in Chapter 4, in which we specify and implement a virtual machine for the programming language, based on the later's formal calculus.

Finally, in Chapter 5 we present the conclusions of this work.

Chapter 2

Programming Sensor Networks

In this chapter we give 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.

2.1 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. We briefly describe each approach, giving examples of programming languages/systems for each layer, and focusing on their advantages and disadvantages.

2.1.1 Binary Code

One of the most widespread programming language for WSN is nesC [4], a C-like language. nesC is coupled with the TinyOS [5] operating system, that was developed for highly constrained embedded systems, such as sensor devices. The core of the system is built around a lightweight event scheduler that execute programmed tasks non-preemptively. nesC applications are built from user and system components. Each component can provide an interface as well has use one. An interface defines a set of functions, that can be associated with commands or events. A function tagged

```
apps/Blink/BlinkC.nc:
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }
  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }
  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

Figure 2.1: An example of nesC programming.

as a command is used to start operations, whilst an event-tagged is used to gather data asynchronously. A component that provides an interface needs to implement the commands defined there. On the other hand, a component that uses such interface, needs to implement the defined events. So, the connection between components by the same interface leads to a two-way data flow.

A code example of nesC language can be seen in Figure 2.1, in which, we can view a component `BlinkC` that uses a sequence of interfaces. The `implementation` code block contains the implementation of the events.

The level of abstraction in this layer provides the developer with a sophisticated control of the sensor's resources (*e.g.* battery, power signal, etc.). Nevertheless, this control does not overshadow the disadvantages that this kind of programming leads to. The most problematic aspect of this type of programming is the limitation of the sensor network's scalability, since sensor nodes have to be programmed individually. There-

fore, the installation of an application on a large sensor network becomes impracticable if further debugging is necessary.

Another model that falls in this level of abstraction is Contiki [6], a lightweight operating system implemented in the C programming language. Contiki provides dynamic loading/unloading of distinct programs and services. When a program calls a service, it uses a *service interface stub*. In the first call the stub looks up the service in a *service layer*, and gets a binding for it. A comparison is done between the version numbers in the service interface and in the interface stub. If a match occurs, the stub calls the implementation of the requested function, which is located in the service process. In Figure 2.2 we can visualize the processing of a service call.

This operating system does not support a multi-threading environment. To add this feature to Contiki systems, Protothreads [7], a lightweight stackless type of threads, was developed. It was designed for systems with severe memory constraints.

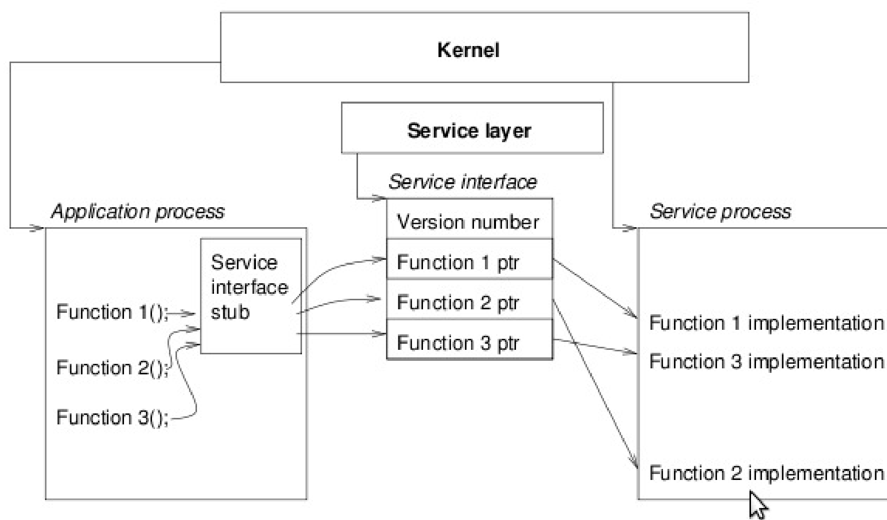


Figure 2.2: An example of a service call in Contiki.

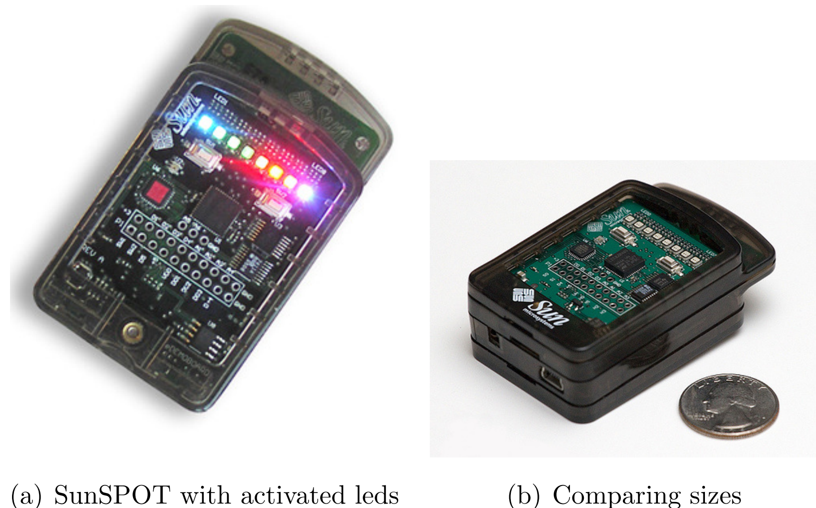
2.1.2 Virtual Machines

At this level, the programmer has at his disposal an infrastructure that permits an abstraction of the underlying software (*e.g.* operating system) and hardware. The infrastructure provides the programmer with fine-grained control of the applications.

There are few virtual machine implementations for sensor networks. One of the

better known in the literature is Maté [8], a *communication-centric* virtual machine. The main focus of this virtual machine is the communication between sensors in the network. Programming sensor networks with Maté can be done using the TinyScript language or using a higher level programming language called Mottle [9]. Programs are called *capsules* and they may be injected in the network, when needed, to achieve specific tasks. 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) [10], used in the Regiment [11] 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. All the execution and communication is based on event handlers. It has an associated intermediate language, called Token Machine Language (TML) [10], that can be targeted by compilers for higher level systems.



(a) SunSPOT with activated leds (b) Comparing sizes

Figure 2.3: An example of a SunSPOT device.

Finally, Sun Microsystems introduced the Squawk virtual machine [12] to support their SunSPOT® devices (Figure 2.3). Squawk is a small and compact Java virtual machine (comparing to Sun's HotSpot virtual machine implementation) written in Java that runs without an underlying operating system on small wireless devices. The project was inspired in the Squeak [13] virtual machine. Squawk's architecture is based on a *split-vm* design with the classfile preprocessor (usually called the *translator*) on one end and the execution engine on the other. The classfile preprocessor produces a

more compact version of the Java byte-code.

Class loading and verification is done at the base stations, which have less resource restrictions. The output produced by the *translator* is a collection of internal class data structures packaged into files called *suites*. These files can form a chain where each following suite only refers to previous suite. Combining this with the object serialization mechanism, the Squawk virtual machine can save a set of loaded and translated classes. The suite files can be sent over a radio link to the devices in the network and, on arrival, de-serialized and interpreted, providing a faster alternative to standard classfile loading. As a downside, the virtual machine startup time is largely augmented.

Portability and security are some of the most interesting features to be gained from the use of virtual machines. On the other hand, they have memory and computational demands that can be incompatible with the most restrictive devices.

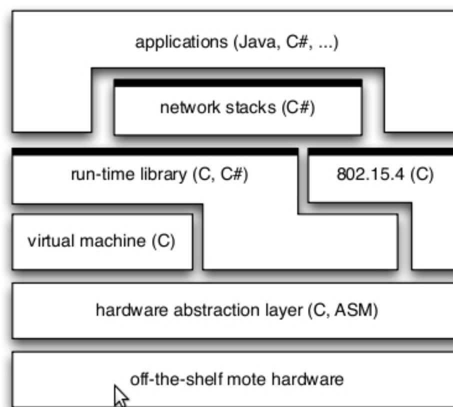


Figure 2.4: IBM Mote Runner architecture.

Another such system is the IBM Mote Runner [14], a run-time environment for wireless sensor networks that consists of a large number of interconnected components. It provides a resource-efficient and high-performance virtual machine that abstracts applications from hardware features. Mote Runner allows applications for WSN to be developed in object-oriented languages such as such as C# and Java. Figures 2.4 and 2.5 present the IBM Mote Runner's architecture and on-mote components, respectively.

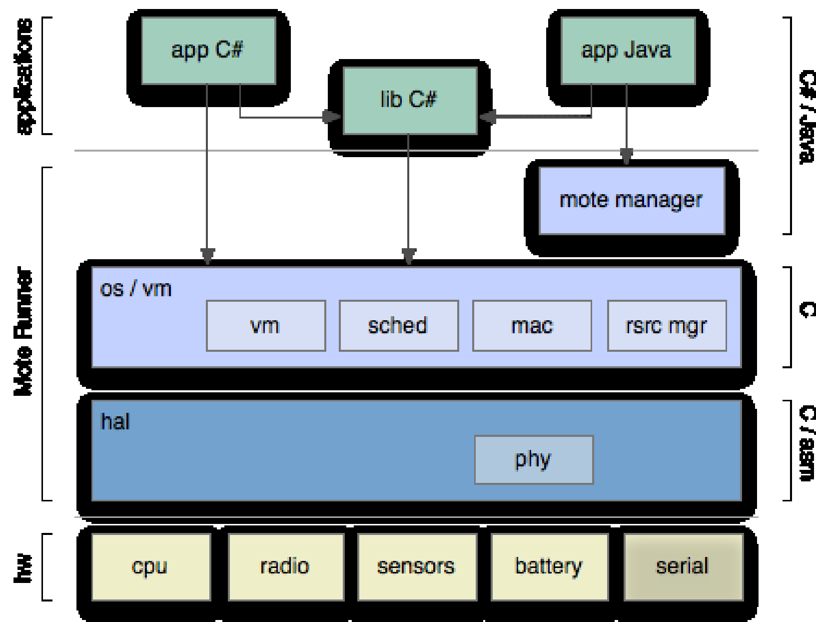


Figure 2.5: IBM Mote Runner components.

2.1.3 Middleware

Another approach to programming WSNs is to use APIs provided by an underlying middleware, which hides the details of the sensor network from the developer. The following discussion is mostly based on [15].

EnviroTrack [16] is an object-based distributed middleware. The programming abstractions are designed to facilitate the development of applications to monitor physical environment. To make this possible, events are labeled in the applications, and are first-class values (*e.g.* a war surveillance application that monitors vehicle movement behind enemy lines can assign labels to individual vehicles according to their battle-state, and be notified when important changes occur).

The complexity of object mobility, communication monitoring, and state of the sensor network is hidden in a library. This system has been implemented on a sensor network platform based on MICA motes. Therefore, Envirotrack's initial applications were compiled to nesC and executed on top of TinyOS.

Impala [17] is a another middleware that has been designed to encourage application modularity, simplicity, adaptivity, and repairability. It is implemented on top of the ZebraNet [17] system, a mobile WSN system proposed to refine monitoring technology.

It uses techniques of store-and-forward communication, and low-power consumption monitoring nodes. ZebraNet's essential concerns go to the deployment, management and communication issues for large numbers of sensors.

2.2 High-level programming

Here the programmer is provided with high-level abstractions for sensors and networks that hide networking and communication details.

At network level, the high-level programming can be divided in macroprogramming or sensor-based programming. The main idea of macroprogramming 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. 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. A compiler or a bundle of run-time libraries should take care of those details. 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.

In systems using the *local behavior* approach, the network is partitioned in regions to specify a computation. Abstract Regions [18], HOOD [19], Regiment [11], Kairos [20] and SNACK [21] are examples of systems and programming languages that are based on a *local behavior* approach. Regiment [11] 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.

On sensor-based programming systems, the responsibility to specify the role of each sensor (implementation, compilation and deployment) belongs to the programmer. He must possess knowledge of the architecture of the sensor network and even perhaps the hardware of each sensor node. SmartMessages [22] is based on this type of approach. SmartMessages'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* [23].

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. 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. Higher level programming in CALLAS is achieved by encoding high-level constructs in the core programming model, thus preserving semantics.

Chapter 3

The Programming Model

Callas [3, 24] 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.

In this chapter we present the Callas programming model which led to the development of the virtual machine presented in this thesis.

3.1 Abstract Syntax

The calculus abstracts away from the physical, link, and network layers of the protocol stack. The focus goes entirely to the challenges of programming sensor network applications, by modeling interactions at the level of the application layer. Figure 3.1 presents the abstract syntax for Callas.

A network is represented as S or as $\mathbf{0}$, being the later the representation of an empty network. 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:

- C - a call-stack for the running process;
- R - a priority queue of runnable processes;
- M - a table with the installed code modules;

- T - a table of timers for function calls;
- I - a queue of incoming messages from the network;
- O - a queue of outgoing messages for the network;
- p - the current position;
- 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 $\langle l(\vec{v})\rangle$. 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 $l = (\vec{x})P$, where l is the name of the function, $\vec{x}$ 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.

A process P can: call a function ($v.l(\vec{v})$), call an external function (**extern** $l(\vec{v})$), install a module (M.**install** M'), send a message (**send** $l(\vec{v})$), receive a message (**receive**), program a timed call (**timer** $l(\vec{v})$ **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** $l(\vec{v})$ 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** $l(\vec{v})$ **every** v **expire** v, the call $l(\vec{v})$ is executed periodically until the timer expires. A timed call that does not expire is written as **timer** $l(\vec{v})$ **every** v **forever**. Finally, the **let** construct permits the processing of intermediate values in computations.

We present a small example that samples the network for data for an interval and with a given frequency. This example has two elements: the code to be run at a base-station - *sink* - and the code to be run at each of the sensors - *sensors*.

$$
\begin{array}{rlll}
S & ::= & & \textit{Sensors} \\
 & & \mathbf{0} & \text{empty network} \\
 & \mid & S \mid S & \text{composition} \\
 & \mid & [C, R \triangleright M, T]^{I,O}_{p,t} & \text{sensor} \\
 & & & \\
M & ::= & & \textit{Modules} \\
 & & \{l_i = (\vec{x}_i)\, P_i\}_{i \in I} & \text{module} \\
 & & & \\
P & ::= & & \textit{Processes} \\
 & & v & \text{value} \\
 & \mid & v.l(\vec{v}) & \text{function call} \\
 & \mid & \textsf{extern}\ l(\vec{v}) & \text{external call} \\
 & \mid & \textsf{timer}\ l(\vec{v})\ \textsf{every}\ v\ \textsf{expire}\ v & \text{timed call} \\
 & \mid & \textsf{send}\ l(\vec{v}) & \text{communication} \\
 & \mid & \textsf{receive} & \text{communication} \\
 & \mid & v.\textsf{install}\ v & \text{install code} \\
 & \mid & \textsf{let}\ x = P\ \textsf{in}\ P & \text{sequence} \\
 & & & \\
v & ::= & & \textit{Values} \\
 & & b & \text{built-in value} \\
 & \mid & x & \text{variable} \\
 & \mid & M & \text{module} \\
 & \mid & \textsf{sensor} & \text{sensor interface} \\
 & & & \\
m & ::= & \langle l(\vec{v}) \rangle & \text{messages} \\
 & & & \\
I, O & ::= & m_1 :: \cdots :: m_n & \text{message queues} \\
R & ::= & P_1 :: \cdots :: P_n & \text{run-queue}
\end{array}
$$

Figure 3.1: The Callas abstract syntax.

```
// sink
install { gather = (self ,x,y) extern log(x,y) };
install { receiver = (self) receive };
timer receiver() every dt forever;
send setup(period , interval)

// sensors
install { setup  = (self ,x,y) timer self.sample() every x expire y
          sample = (self)      let x = extern time() in
                               let y = extern data() in
                               send gather(x,y)               };
install { receiver = (self) receive };
timer receiver() every dt forever
```

The first step taken by this program is installing a module that contains the function gather in the sink interface (M). This function logs the arguments using an **extern** call. Next, it broadcasts a call to setup with the period for the call being triggered, and the time interval for the sampling process. This call is then placed in the output queue (O) of the sink. All the message routing is managed at the network layer and below (see next section). Both sink and sensor install a function receiver in order to synchronize and receive values from the network. Each sensor, that has an installed version of the setup function, receives the messages transmitted by the sink. The information generated by the sensors, in the form of gather messages, is transmitted to the sink due to the nonexistence of gather functions in sensors.

The functions setup and sample are installed in each sampling sensor in the network. If a call to setup arrives from the network, through I, the sensor launch a timer to periodically call the sample function. With the periodic execution of this function, the local time and the requested data are read with **extern** calls and sent, in the form of a gather message, to the network.

3.2 Semantics

The operational semantics is defined with the help of a structural congruence as usually in process calculi [25], the congruence rules are given in Figure 3.2. The only non-standard rule is $[C, R \triangleright M, T]_{p,t}^{I,O} \equiv [C, R \triangleright M, T]_{p,t}^{I,O}\{\mathbf{0}\}$, 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.

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-

$$S_1 \,|\, S_2 \equiv S_2 \,|\, S_1, \qquad S \,|\, \mathbf{0} \equiv S, \qquad S_1 \,|\, (S_2 \,|\, S_3) \equiv (S_1 \,|\, S_2) \,|\, S_3 \qquad \text{(S-MONOID-SENSOR)}$$

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

Figure 3.2: Structural congruence for sensors.

MODULE. 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.

$$\frac{\text{noEvent}(T, t)}{[\mathcal{C}[\![\textbf{sensor}.\textbf{install}\ \ M']\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [\mathcal{C}[\![\{\}]\!] : C, R \triangleright M + M', T]^{I,O}_{p,t+1}} \qquad \text{(R-INSTALL-INTERFACE)}$$

$$\frac{\text{noEvent}(T, t)}{[\mathcal{C}[\![M_1.\textbf{install}\ \ M_2]\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [\mathcal{C}[\![M_1 + M_2]\!] : C, R \triangleright M, T]^{I,O}_{p,t+1}} \qquad \text{(R-INSTALL-MODULE)}$$

If a timed call is fired, it is placed in the priority queue, it does not preempt the current process. The *noEvent* predicate is responsible for monitoring the time of the next activation for every timed call, and comparing it to the current time.

$$\frac{T' = T \uplus (l(\vec{v}), v_1, t + v_2, t + v_1) \quad \text{noEvent}(T, t)}{[\mathcal{C}[\![\textbf{timer}\ \ l(\vec{v})\ \textbf{every}\ \ v_1\ \textbf{expire}\ \ v_2]\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [C, \mathcal{C}[\![\textbf{sensor}.l(\vec{v})]\!] :: R \triangleright M, T']^{I,O}_{p,t+1}} \qquad \text{(R-TIMER)}$$

$$\frac{t \leq v_2 \quad T' = T \uplus (l(\vec{v}), v_1, v_2, t + v_1)}{[C, R \triangleright M, T \uplus (l(\vec{v}), v_1, v_2, t)]^{I,O}_{p,t} \rightarrow [C, \textbf{sensor}.l(\vec{v}) :: R \triangleright M, T']^{I,O}_{p,t}} \qquad \text{(R-TIMER)}$$

$$\frac{t > v_2}{[C, R \triangleright M, T \uplus (l(\vec{v}), v_1, v_2, t)]^{I,O}_{p,t} \rightarrow [C, R \triangleright M, T]^{I,O}_{p,t}} \qquad \text{(R-EXPIRE)}$$

This firing is achieved when the adequate t value is reached and is controlled by the rules R-TIMER and R-EXPIRE. The R-TIMER rule puts the timed function call $l(\vec{v})$ at the front of the run-queue. On the other hand, the R-EXPIRE rule removes the corresponding tuple from T when the timer has expired. All the timed-calls must

be made to functions already installed in M. Other calls are neglected by the rule R-NO-FUNCTION, whilst the call's execution is delegated to the rule R-CALL-INTERFACE.

$$\frac{l \notin \mathrm{dom}(M) \quad \mathrm{noEvent}(T,t)}{[\mathcal{C}[\![\textsf{sensor}\,.l(\vec{v})]\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [\mathcal{C}[\![\{\}]\!] : C, R \triangleright M, T]^{I,O}_{p,t+1}} \quad \text{(R-NO-FUNCTION)}$$

$$\frac{M(l) = (\textsf{self}\ \ \vec{x})P \quad \mathrm{noEvent}(T,t)}{[\mathcal{C}[\![\textsf{sensor}\,.l(\vec{v})]\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [\mathcal{C}[\![P[\textsf{sensor}\,\vec{v}/\textsf{self}\ \ \vec{x}]]\!] : C, R \triangleright M, T]^{I,O}_{p,t+1}}$$
(R-CALL-INTERFACE)

All the sensor's communication with the network is controlled by the rules R-SEND and R-RECEIVE.

$$\frac{\mathrm{noEvent}(T,t)}{[\mathcal{C}[\![\textsf{send}\ \ l(\vec{v})]\!] : C, R \triangleright M, T]^{I,O}_{p,t} \rightarrow [\mathcal{C}[\![\{\}]\!] : C, R \triangleright M, T]^{I,O::\langle l(\vec{v})\rangle}_{p,t+1}} \quad \text{(R-SEND)}$$

$$\frac{\mathrm{noEvent}(T,t)}{[\mathcal{C}[\![\textsf{receive}\,]\!] : C, R \triangleright M, T]^{\langle l(\vec{v})\rangle::I,O}_{p,t} \rightarrow [C, R :: \textsf{sensor}\,.l(\vec{v}) \triangleright M, T]^{I,O}_{p,t+1}} \quad \text{(R-RECEIVE)}$$

The network's reduction semantics is carried out combined with in-sensor processing. The reduction handles the distribution of messages placed by the sensors in their output queues (O) [3]. The *inRange* predicate checks whether a sensor is within communication range of the current broadcasting sensor. A transmission starts with the structural congruence rule (Figure 3.2) S-INIT-SEND, applies R-BROADCAST multiple times, and terminates with an application of R-RELEASE. The *networkRoute* predicate simply forwards the messages to I,O.

$$\frac{\mathrm{inRange}(p,p') \qquad (I'',O'') = \mathrm{networkRoute}(m,I',O')}{[C, R \triangleright M, T]^{I,m::O}_{p,t}\{S\} \mid [R' \triangleright M', T']^{I',O'}_{p',t'} \rightarrow [C, R \triangleright M, T]^{I,m::O}_{p,t}\{S \mid [R' \triangleright M', T']^{I'',O''}_{p',t'}\}}$$
(R-BROADCAST)

$$[C, R \triangleright M, T]^{I,m::O}_{p,t}\{S\} \rightarrow [C, R \triangleright M, T]^{I,O}_{p,t} \mid S \quad \text{(R-RELEASE)}$$

There is a type system addressed to the calculus. This reduction semantics preserves the types, thus the types are invariant. The calculus coupled with the type system, achieve the *type-safety* property [3].

3.3 Language Syntax

We present 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 [26]. 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.

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

Figure 3.3: Sampling program - Sink.

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

Figure 3.4: Sampling program - Sensor.

In Figures 3.3 and 3.4 we present the sampling program (see before in section 3.1). In this example, the first delimited block starts with the keyword **sensor** and its only constituent is the keyword **run**. The later marks the initial process, while the **sensor** is a marker for a type of sensor. Now, we relate the constructs of the sample function with the correspondent abstract syntax:

```
sample = (self)    let x = extern time() in
                   let y = extern data() in
                   send gather(x,y)
```

We can see differences in the definition of the function sample, which in the Callas language is written with the **def** keyword, just like Python does. The scope of the function extends to the blocks with same indentation. Also, to simplify the language, the **let** construct disappears in Callas.

The expression **module** Sampling **as** sampling assigns the declared module of type Sampling to variable sampling. This declaration is just a syntactic sugar to link types (such as Sampling) to variables (such as sampling)

```
# Sink
  run:
    module Ping as ping:
      def forward(self, thn, mac):
        now = extern time()
        elapsed = now - thn
        extern log(elapsed, mac)
      def receiver(self):
            receive
    sensor.install(ping)
    send ping()
    fire receiver() every dt forever
```

Figure 3.5: Ping program - Sink.

```
# Sensor
  run:
    module Ping as ping:
      def ping(self):
        time = extern time()
        mac  = extern mac()
        send forward(time, mac)
      def receiver(self):
            receive
    sensor.install(ping)
    fire receiver() every dt forever
```

Figure 3.6: Ping program - Sensor.

In Figures 3.5 and 3.6 we can see the Ping example, in which, sensors within a network of sensors send simultaneously their MAC addresses and the current time to the

network, while a sink sensor logs the elapsed time along with the MAC addresses as it receives messages from the network's sensors. In this example, each sensor has its own function ping and simply waits for remote calls from the sink. Also, we can denote the **fire** receiver () **every** dt **forever** construct, that maintains the same syntax as in the abstract syntax. This construct is also present in the other examples.

Finally, we focus on a more complex example that gathers the maximum value of a data attribute and the correspondent sensor's MAC address (Figure 3.7). In this program, the sink contains a function gather, similar to the one in the sampling program, and has a remote call to the function setup.

On the other hand, each sensor has two functions defined: a different version of the gather function, and the setup function. The later function computes some data and its mac and forwards those values to the network. The gather compares the maximum value between the value received with the one computed by itself. The resulting value is sent to the network. This particular example makes use of conditional statements not present in the base calculus but that can be easily included. We can see that use in the following code snippet:

```
if x > val:
          module Value as value:
            def max_data():
              return x
            def max_mac():
              return y
          sensor.install(value)
        elif x < val:
          # do something else
        else:
          pass
```

Finally, we can denote that both defined functions in this snippet make use of the **return** instruction.

A compiler with statically type verification was created for Callas language. This verification grants the nonexistence of protocolar run-time errors, such as: non-module method invocation, method invocation with wrong number/type of parameters. Further, it generates byte-code for the virtual machine, which is introduced in the Chapter 4.

```
#sink
  run:
    module Maximum as maximum:
      def gather(self, x, y):
        extern log(x,y)
      def receiver(self):
        receive
    sensor.install(maximum)
    send setup()
    timer receiver() every dt forever

#sensor
  run:
    module Maximum as maximum:
      def setup(self):
        x = extern data()
        y = extern mac()
        module Value as value:
           def max_data(self):
             return x
           def max_mac(self):
             return y
        sensor.install(value)
        send gather(x,y)

      def gather(self, x, y):
        val = max_data()
        if x > val:
          module Value as value:
            def max_data():
              return x
            def max_mac():
              return y
          sensor.install(value)
        elif x < val:
          # do something else
        else:
          pass
        send gather(x,y)
      def receiver(self):
        receive
    sensor.install(maximum)
    timer receiver() every dt forever
```

Figure 3.7: Maximum value of data attribute example - Sensor.

Chapter 4

The Virtual Machine

After having explored the programming model in the previous chapter, here we present the main contribution of this dissertation. Our goal is to design from scratch a virtual machine with an operational semantics inspired on that from the calculus [3].

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® platform for the development. The SunSPOT® devices run a compact version of a Java virtual machine, called Squawk.

4.1 Format of byte-code

In the Figure 4.1 we present the byte-code format that the Callas compiler generates.

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.

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. All the instructions are given by c, and each constant symbol is given by u, that can be one of the follow: $\mathsf{BOOL}\ k$, $\mathsf{INTEGER}\ n$, $\mathsf{FLOAT}\ e$, $\mathsf{STRING}\ k\ l$,

tags (1 byte)		MAGIC | VERSION |
		BOOL | INTEGER | STRING | FLOAT |
		MODULES | MODULE | SYMBOLS | LOCALS |
		TYPES | CODE | FUNCTION | OPEN
short integers (1 byte)		k
integers (4 bytes)		n
floats (4 bytes)		e
strings (≤ 256 bytes)		l
symbols	u ::=	BOOL k | INTEGER n | FLOAT e | STRING k l | m
instructions (1 or 2 bytes)	c ::=	
build and install modules		loadm k | install |
calls		extern | call | timer |
control flow		goto n | return | halt |
networking		receive | send |
data movement		loadb | loadc k | load k |
		storeb | store k | dup | swap |
arithmetic		add | sub | mul | div | mod |
logical		not | and | or |
relational		eq | neq | gt |
		lt | leq | geq |
		if_true n
function	f ::=	FUNCTION STRING k l LOCALS k CODE n $\vec{c}$ SYMBOLS k $\vec{u}$
module	m ::=	MODULE k $\vec{f}$ OPEN k
program	p ::=	MAGIC n VERSION k MODULES k $\vec{m}$ TYPES k $\vec{u}$

Figure 4.1: The byte-code format.

or a module m. Each one of these sections is marked by $tags$.

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

The notation for the frames, stacks, and queues of the virtual machine can be seen in Figure 4.3. Frames with extra, uninitialized slots are used to accommodate both parameters and the local variables of functions.

In Figure 4.4, we define the initial machine state. The incoming-, outgoing-, and run-queues are empty. The table of timed calls is also empty, along with the sensor's interface, which still has no code installed. The call-stack contains a the tuple with an empty operand-stack, an empty frame, the byte-code corresponding to the "`run`" function, and a program counter with value zero.

4.3 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 Chapter 3, and is parametric in the program p.

byte array	$b \in$	$\mathcal{B}$
word	$w \in$	$\mathcal{W} = \mathsf{Bool} \cup \mathsf{Int} \cup \mathsf{Float} \cup \mathsf{String} \cup \mathcal{M}$
frame		$\mathcal{F} = \langle \vec{\mathcal{W}} \rangle$
operand-stack		$\mathcal{S} = \vec{\mathcal{W}}$
timer		$\mathcal{T} = \mathcal{S} \times \mathsf{Int} \times \mathsf{Int} \times \mathsf{Int}$
incoming-queue		$\mathcal{I} = \vec{\mathcal{F}}$
outgoing-queue		$\mathcal{O} = \vec{\mathcal{F}}$
runnable process		$\mathcal{H} = \mathcal{S} \times \mathcal{B}$
run-queue		$\mathcal{R} = \vec{\mathcal{H}}$
running process		$\mathcal{G} = \mathcal{S} \times \mathcal{F} \times \mathsf{Int} \times \mathcal{B}$
call-stack		$\mathcal{C} = \vec{\mathcal{G}}$
module	$m \in$	$\mathcal{M} = \mathsf{String} \mapsto \mathcal{B}$
machine state		$\mathsf{Int} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{I} \times \mathcal{O} \cup \{\mathsf{halt}\}$

Figure 4.2: The syntactic categories of the virtual machine.

n-initialized k-extra frame	$\langle \vec{w} \rangle_k \equiv \langle w_1, \ldots, w_n, 0_1, \ldots, 0_k \rangle, k \geq 0$
frame	$\langle \vec{w} \rangle \equiv \langle \vec{w} \rangle_0$
stacks of syntactic category α	$\alpha_1 : \cdots : \alpha_n \mid \epsilon$
queues of syntactic category α	$\alpha_1 :: \cdots :: \alpha_n \mid \epsilon$

Figure 4.3: Notation for the components of the virtual machine.

$$\langle 0, \emptyset, \emptyset, (\epsilon, \epsilon, 0, b), \epsilon \rangle_{\epsilon}^{\epsilon}$$

where the program p, the initial module m and the byte-code b for the `"run"` function are obtained from the byte-code as follows:

$$\begin{aligned} p &= \mathit{loadProgram}(\mathsf{MAGIC}\ n\ \mathsf{VERSION}\ k\ \mathsf{MODULES}\ k\ \vec{m}\ \mathsf{TYPES}\ k\ \vec{u}) \\ m &= \mathit{lookupModule}(p, 0) \\ b &= \mathit{lookupFunction}(m, "\mathtt{run}") \end{aligned}$$

Figure 4.4: The initial state.

In the sequel we overview the main computational rules. The generic form of the rules is:

$$\frac{\mathit{pre} - \mathit{conditions}}{\mathit{original\ state} \to \mathit{new\ state}}$$

Beginning with the install instruction, it looks at the top of the operand-stack ($\mathcal{S}$), takes two modules and merges them, putting the result at the top of the operand-stack.

$$\frac{\mathcal{B}[j] = \mathsf{install}}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (m : m' : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (m + m' : \mathcal{S}, \langle \vec{w} \rangle, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}\end{array}}$$

The call instruction expects that the operand-stack is a sequence of the form $m : l : k : w_1 : \cdots : w_k : \mathcal{S}$, in which the module m and the function label l are used to lookup the function's byte-code ($\mathcal{B}'$) to be executed. The next words in operand-stack are the parameters to the function. These parameters are copied to the environment frame of a new tuple, placed at the top of the call-stack. This frame has a fixed size which is determined by the number of parameters and locals of the function ($k + k'$). Besides this environment frame, the new tuple also has an empty operand-stack, a program counter initialized with zero and the byte-code $\mathcal{B}'$ to run.

$$\frac{\mathcal{B}[j] = \mathsf{call} \qquad \mathcal{B}' = \mathit{lookupFunction}(m, l) \qquad k' = \mathit{numberLocals}(\mathcal{B}')}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (m : l : k : w_1 : \cdots : w_k : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\mathcal{I}} \rightarrow \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (\epsilon, \langle \vec{w} \rangle_{k+k'}, 0, \mathcal{B}') : (\mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\mathcal{I}}\end{array}}$$

Synchronous external calls can be performed with the instruction extern. Here, the operand-stack has a similar configuration that in the call instruction, except for the module at the top. These calls depend on the device where the virtual machine runs, more specifically on the operation system or middleware in which the virtual machine is embedded. The result of the call is placed at the top of the operand stack.

$$\frac{\mathcal{B}[j] = \mathsf{extern} \qquad w = \mathit{callExtern}(l : k : w_1 : \cdots : w_k)}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (l : k : w_1 : \cdots : w_k : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\mathcal{I}} \rightarrow \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\mathcal{I}}\end{array}}$$

The instructions send and receive handle messages to/from the network.

In the send instruction, the operand-stack has the same configuration as in an extern instruction. A frame with all the elements that are at the top of this stack $\mathcal{S}$ is produced and placed at the outgoing queue ($\mathcal{O}$).

$$\frac{\mathcal{B}[j] = \mathsf{send}}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (l : k : w_1 : \cdots : w_k : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\mathcal{I}} \rightarrow \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O} :: \langle w_1, \ldots, w_k, l \rangle}^{\mathcal{I}}\end{array}}$$

The receive takes a frame from the incoming-queue ($\mathcal{I}$) and puts a new, low-priority, runnable process in the run-queue ($\mathcal{R}$). Note that the sensor interface $\mathcal{M}$, is also placed at the top of the new operand-stack, as the remote call, must address one of the installed functions of the sensor.

$$\frac{\mathcal{B}[j] = \mathsf{receive}}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (_, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle_{\mathcal{O}}^{\langle w_1, \ldots, w_k, l \rangle :: \mathcal{I}} \rightarrow \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (_, _, j+1, \mathcal{B}) : \mathcal{C}, R :: (\mathcal{M} : l : k : w_1 : \cdots : w_k, \mathsf{call})\rangle_{\mathcal{O}}^{\mathcal{I}}\end{array}}$$

The following rules handles timed calls. The first one corresponds to the creation of a timed call, which is done by the timer instruction. This instruction expects that

function name and arguments of the call, plus the sensor interface and two integers, to be at the top of the operand-stack. These integers correspond to the period and the expire time of the call. The rule puts such a tuple in the timer table ($\mathcal{T}$). The triggering of this timed call is handled by another rule, which creates a corresponding high priority runnable process and puts it in $\mathcal{R}$. When the timed call expires, another rule discards it by eliminating the corresponding entry in $\mathcal{T}$.

$$\frac{\mathcal{B}[j] = \mathsf{timer} \qquad S' = \mathcal{M} : l : k : w_1 : \cdots : w_k}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (n_1 : n_2 : S' : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \\ \qquad \langle t+1, \mathcal{M}, \mathcal{T} + \{(S', n_2, t + n_2, t + n_1)\}, (\mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}}$$

$$\frac{t_1 \leq t \leq t_2 \qquad S = \mathcal{M} : l : k : w_1 : \cdots : w_k}{\langle t, \mathcal{M}, \mathcal{T}.\{(S, n_1, t_1, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t, \mathcal{M}, \mathcal{T}.\{(S, n_1, t + n_1, t_2)\}, \mathcal{C}, (S, \mathsf{call}) :: \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{t > t_2 \qquad S = \mathcal{M} : l : k : w_1 : \cdots : w_k}{\langle t, \mathcal{M}, \mathcal{T}.\{(S, n_1, t_1, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

When the execution of a function ends with a return and has a value to return, the value is placed at the top of the operand-stack in the previous tuple of the call-stack. If the function has no value to return, the previous tuple resumes its execution without any change.

$$\frac{\mathcal{B}[j] = \mathsf{return}}{\langle t, \mathcal{M}, \mathcal{T}, (w, _, j, \mathcal{B}) : (S', _, _, _) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (w : S', _, _, _) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{return}}{\langle t, \mathcal{M}, \mathcal{T}, (\epsilon, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, \epsilon, (\mathcal{M} : l : k : w_1 : \cdots : w_k, \mathsf{call}) :: R\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \\ \qquad \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{M} : l : k : w_1 : \cdots : w_k, \epsilon, _, \mathsf{call}), \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}$$

If the $\mathcal{C}$ is empty, the machine takes a tuple from R and moves it to $\mathcal{C}$. The execution of the virtual machine ends whenever the halt instruction is executed. If the virtual machine does not have any instruction to execute, and both run-queue and call-stack are empty, it simply continues its execution with no change.

$$\langle t, \mathcal{M}, \mathcal{T}, \epsilon, \epsilon \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, \mathcal{M}, \mathcal{T}, \epsilon, \epsilon \rangle_{\mathcal{O}}^{\mathcal{I}}$$

$$\frac{\mathcal{B}[j] = \mathsf{halt}}{\langle t, \mathcal{M}, \mathcal{T}, (_, _, j, \mathcal{B}) : C, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \mathsf{halt}}$$

The instructions loadb and storeb are used to load and store the sensor interface module $\mathcal{M}$. The instruction loadm k loads a generic module where k represents the relative pointer of the module in the program p. The module is placed at the top of the operand-stack.

$$\frac{\mathcal{B}[j] = \mathsf{loadb}}{\langle t, \mathcal{M}, \mathcal{T}, (\mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{M} : \mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{storeb}}{\langle t, \mathcal{M}, \mathcal{T}, (m : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, m, \mathcal{T}, (\mathcal{S}, _, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{loadm}\ k \qquad m = loadModule(p, k) \qquad k' = numberFunctions(m)}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (n_1 : v_{11} : \cdots : v_{n1} : \cdots : n'_k : v_{1k'} : \cdots : v_{nk'} : \mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (m : \mathcal{S}, _, j+2, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}\end{array}}$$

Moving values between the environment frame and the operand-stack is done using load k and store k. When we need to load a constant kept in the symbols block of the byte-code for a function to the operand-stack, we use the loadc k instruction, where k is the relative pointer of the symbol in the byte-code.

$$\frac{\mathcal{B}[j] = \mathsf{loadc}\ k \qquad w = lookupSymbol(\mathcal{B}, k)}{\langle t, \mathcal{M}, \mathcal{T}, (\mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, _, j+2, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{load}\ k \qquad \langle \vec{w} \rangle = \langle w_1 \ldots w_k \ldots w_{m+k} \rangle}{\langle t, \mathcal{M}, \mathcal{T}, (\mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, \mathcal{M}, \mathcal{T}, (w_k : \mathcal{S}, \langle \vec{w} \rangle, j+2, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{store}\ k \qquad \langle \vec{w} \rangle = \langle w_1 \ldots w_k \ldots w_{m+k} \rangle}{\langle t, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}} \to \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{S}, \langle w_1 \ldots w \ldots w_{m+k} \rangle, j+2, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}}$$

Other instructions such as dup, and swap were also defined. Their definition is based on similar instructions in the Java virtual machine.

$$\frac{\mathcal{B}[j] = \mathsf{dup}}{\langle t, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (w : w : \mathcal{S}, \langle \vec{w} \rangle, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{swap}}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (w_1 : w_2 : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \\ \quad \langle t+1, \mathcal{M}, \mathcal{T}, (w_2 : w_1 : \mathcal{S}, \langle \vec{w} \rangle, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}}$$

Basic arithmetic and logic operations, are also supported. The evaluation of the values, situated at the top of the operand-stack, is placed at the top of the operand-stack. Unary operations are similar.

$$\frac{\mathcal{B}[j] = \mathit{binop} \qquad w = w_1 \, \mathit{binop} \, w_2}{\langle t, \mathcal{M}, \mathcal{T}, (w_1 : w_2 : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, \langle \vec{w} \rangle, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \mathit{unop} \qquad w = \mathit{unop} \, w_1}{\langle t, \mathcal{M}, \mathcal{T}, (w_1 : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (w : \mathcal{S}, \langle \vec{w} \rangle, j+1, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

We define simple control flow instructions with goto and if_true n with the usual semantics.

$$\frac{\mathcal{B}[j] = \mathsf{goto}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (\mathcal{S}, _, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{S}, _, n, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{if_true}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (\mathsf{True} : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{S}, \langle \vec{w} \rangle, j+n, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \mathsf{if_true}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (\mathsf{False} : \mathcal{S}, \langle \vec{w} \rangle, j, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow \langle t+1, \mathcal{M}, \mathcal{T}, (\mathcal{S}, \langle \vec{w} \rangle, j+5, \mathcal{B}) : \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}}$$

After this overview of the virtual machine specification we now describe in the implementation in detail.

4.4 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® devices. The source code of the virtual machine is provided with this thesis in Appendix A.

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

Table 4.1: Data-structures map.

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  types
                byte[]  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. To be more conservative in terms of memory we use a vector instead of a hashtable, we consider that we do not have a large number of functions in a module. In future work, the use of a hashtable will be preferable.

Class Function has the following attributes: the name, the number of locals, the byte-code, 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. We extract the symbols associated with every function from its byte-code 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.

```
class 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 $\mathcal{H}$ for a given function and environment frame.

The class RunnableProcess implements tuples of set $\mathcal{H}$, that is, tuples containing: an operand-stack, and a byte-code array. An operand-stack comprises the same types that the function's symbols.

```
class RunnableProcess {

        private Vector symbols;
        private byte[] byteCode;
        private Stack  operandStack;
        private int    numberOfLocals;

        RunnableProcess(Function function) { ... }
}
```

Class RunningProcess implements tuples of set $\mathcal{G}$, that is, tuples containing: an operand-stack, an environment frame, a program counter, and a byte-code array. We can view its definition above.

```
class RunningProcess {

        private Vector symbols;
        private Vector environment;
        private Stack  operandStack;
        private int    programCounter;
        private byte[] byteCode;

        RunningProcess(RunnableProcess runnableProcess) { ...   }
}
```

The run-queue is composed by runnable-processes, and the call-stack contains the running-process. The incoming and outgoing queues are represented by the **Queue** class from Java.

We are left with the table of timers and the interface of the sensor. The sensor's interface is a module, so its structure is already defined in the class Module. Next, we present the constructor of class TimedCall.

```
class TimedCall extends Thread {

        private Module brane;
        private String functionName;
        private Vector parameters;
        private int    period;
        private int    expire;
        private static final int ONE_SECOND = 1000;

        TimedCall(Module sensorInterface, String functionName, Vector parameters,
                  int period, int expire) { ... }

        ...
}
```

The constructor receives as parameters, the interface of the sensor, the function's

name, parameters for the function, the period, and the expire time of the call. Note that, the class extends java.lang.Thread and so it must implement a run method. In this particular case this method is used to program the behavior of each timed-call. Next, the run method is shown.

```
public void run() {

  VirtualMachine vm = VirtualMachine.getVirtualMachine();

  int i = 1;
  while (true) {

    Utils.sleep(this.period*TimedCall.ONE_SECOND);

    if (this.expire <= (this.period*i)) {
      break;
    }
    i++;

    RunnableProcess newCall= new RunnableProcess (
              vm.searchFunctionInModule(sensorInterface, functionName));
    int parSize = parameters.size();
    for(int j=parSize;j<parSize;j--){
      newCall.getOperandStack().push(parameters.elementAt(j));
    }

    vm.getRunQueue().put(newCall);
  }
}
```

First, we must get a handle for the run-queue of the virtual machine to place the runnable-processes generated by the thread. Next, we configure the call to sleep the given period between each call. Finally, the call in the loop places the timed-call in the run-queue of the virtual machine.

There are two interfaces defined in the virtual machine, the Opcodes and Tags. The first one keeps the opcodes for each machine instruction. The second one defines the tags that occur in the byte-code, which are used to mark the beginning of events and sections in the former.

The SunSPOT® starts its execution with the function startApp() defined in the class VirtualMachine. This function extracts the program information from the byte array given as parameter and a VirtualMachine object is created (can be seen in Appendix A).

This class defines three threads: mainLoopThread, incomingThread, and outgoingThread.

The mainLoopThread takes care of all the byte-code execution. The incomingThread runs the code that receives messages from the network and de-serializes them. Finally,

the outgoingThread serializes messages and sends them over the network.

When a virtual machine starts the following structures are created and initialized: the run-queue, the incoming-queue, the outgoing-queue, the call-tack, the sensor interface, and the timer table. Next, we can view the initialization of these structures, and a skeleton of the mainLoopThread.

```
public VirtualMachine() throws MalformedByteCodeException{

  runQueue        = new Queue();
  incomingQueue   = new Queue();
  outgoingQueue   = new Queue();
  callStack       = new Stack();
  sensorInterface = new Module();
  timers          = new Vector();

  mainLoopThread = new Thread(new Runnable() {
    public void run() {
      ...
      while (true) {
        ...
        switch (runningProcess.byteCode
               [runningProcess.programCounter]) {
          ...
        }
      }
    }
  });
}
```

As mentioned before, the byte-code interpreter is located in the mainLoopThread. It is also important to detail the implementation of some instructions. The code for the integer binary operations is presented above. The code for the binary and unary operations on the other built-in types are also standard.

```
case Opcodes.IADD:

        int iaddOperand1 = ((Integer) runningProcess.operandStack.pop()).intValue();
        int iaddOperand2 = ((Integer) runningProcess.operandStack.pop()).intValue();
        runningProcess.operandStack.push(iaddOperand1 + iaddOperand2);
        runningProcess.programCounter += 1;
        break;
```

As a result of the relatively large number of opcodes, we focus on the implementation of the most important instructions.

The loadm instruction loads a generic module from the program's byte-code and places

a reference for it at the top of the operand-stack.

The loadb and storeb instructions take care of the operations of load and store the interface of the sensor. The implementations of these instructions can be seen next.

```
case Opcodes.LOADM:
        Module moduleLoaded = (Module) program
                                                  .getModuleAt(runningProcess
                                                               .getByteAt(runningProcess
                                                               .getProgramCounter() + 1));

        if (moduleLoaded.isOpen() == 1) {
          int loadmNumberFunctions = moduleLoaded
          .getNumberOfFunctions();
          int loadmFreeVar;
          for (int loadmIndex = 0; loadmIndex < loadmNumberFunctions; loadmIndex++) {
            Function function = (Function) moduleLoaded
            .getFunctionAt(loadmIndex);
            loadmFreeVar = ((Integer) runningProcess.pop())
            .intValue();
            if (loadmFreeVar > 0) {
              for (int loadmIndex2 = 0; loadmIndex2 < loadmFreeVar; loadmIndex2++) {
                function
                .addSymbol(runningProcess.pop());
              }
              moduleLoaded.setFunctionAt(function,
              loadmIndex);
            }

          }
        }
        runningProcess.push(moduleLoaded);
        runningProcess.setProgramCounter(2);

        break;

case Opcodes.LOADB:
        runningProcess.push(sensorInterface);
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.STOREB:
        sensorInterface = (Module) runningProcess.pop();
        runningProcess.setProgramCounter(1);
        break;
```

As we mentioned, the install instruction takes two modules whose references are placed at the top of the operand-stack, installs the functions from the second module into the first module, and leaves a reference for the resulting module at the top of the operand-stack.

Looking at the code of install, we can see that the two modules that are at the top of the operand-stack, original and toInstall, are merged together, and the result is

the new module.

```
case Opcodes.INSTALL:
        Module originalModule = (Module) runningProcess.pop();
        Module toInstallModule = (Module) runningProcess.pop();
        Enumeration toInstallElements = toInstallModule
                .getFunctions().elements();
        while (toInstallElements.hasMoreElements()) {
                Enumeration originalElements = originalModule
                                                  .getFunctions().elements();
                Function toInstallFunction = (Function) toInstallElements
                                                          .nextElement();
                while (originalElements.hasMoreElements()) {
                Function originalFunction = (Function) originalElements
                                                  .nextElement();
                if (toInstallFunction.getName().equals(
                                        originalFunction.getName())) {
                        int functionIndex = originalModule
                                        .getFunctions().indexOf(
                                                originalFunction);
                        originalModule.setFunctionAt(
                                        toInstallFunction, functionIndex);
                        toInstallFunction = null;
                        break;
                }
        }
        if (toInstallFunction != null) {
                originalModule.getFunctions().addElement(
                                        toInstallFunction);
        }}
        runningProcess.push(originalModule);
        runningProcess.setProgramCounter(1);
        break;
```

Now, we focus on the instruction call for calling a function in a module. This instruction takes the two elements that are at the top of the operand-stack, first one is the module, and the second one the function name. The next elements in the operand-stack are the parameters of the function. The function must be member of the module to the call be made.

```
case Opcodes.CALL:
        Module callBrane = (Module) runningProcess.pop();
        String callFunctionName = (String) runningProcess.pop();
        Function callFunction = callBrane
                        .getFunctionByName(callFunctionName);
        int callNumberParameters = ((Integer) runningProcess
                        .pop()).intValue();
        Vector callParameters = new Vector();
        for (int callIndex = 0; callIndex < callNumberParameters; callIndex++) {
                callParameters.addElement(runningProcess.pop());
        }
        runningProcess.setProgramCounter(1);
        RunnableProcess callRunnableProcess = new RunnableProcess(
                                        callFunction);
```

```
        for (int callIndex = 0; callIndex < callNumberParameters; callIndex++) {
                callRunnableProcess.push(callParameters
                                .elementAt(callIndex));
        }
        runningProcess = new RunningProcess(callRunnableProcess);
        callStack.push(runningProcess);
        break;
```

The timer instruction is responsible for the timed-calls. It has a sequence of parameters, such as: period time, expiring time of the call, the interface of the sensor, the name of the function, and its parameters. All the timed calls are TimedCall objects, and the timers data-structure keeps track of all the calls.

```
case Opcodes.TIMER:
        int timerExpire = ((Integer) runningProcess.pop())
                                        .intValue();
        int timerPeriod = ((Integer) runningProcess.pop())
                                        .intValue();
        Module timerBrane = (Module) runningProcess.pop();
        String timerFunctionName = (String) runningProcess
                                        .pop();
        int numberOfParameters = ((Integer) runningProcess
                                        .pop()).intValue();
        Vector timerParameters = new Vector();
        if (numberOfParameters > 0) {
                for (int timerIndex = 0; timerIndex < numberOfParameters; timerIndex++) {
                        timerParameters.addElement(runningProcess.pop());
                }
        }
        runningProcess.setProgramCounter(1);
        if (timers.isEmpty()) {
                TimedCall timedCall = new TimedCall(timerBrane,
                        timerFunctionName, timerParameters,
                        timerPeriod, timerExpire);
                timers.addElement(timedCall);
                timedCall.start();
                } else {
                        Enumeration timersElements = timers.elements();
                        TimedCall element;
                        while (timersElements.hasMoreElements()) {
                                element = (TimedCall) timersElements
                                        .nextElement();
                                if (timerFunctionName.equals(element
                                                        .getFunctionName())) {
                                        // Exception
                                }
                        }
                        timers.addElement(new TimedCall(timerBrane,
                                timerFunctionName, timerParameters,
                                timerPeriod, timerExpire));
                }
        break;
```

Next, we present the network's related instructions, the send and receive .

The send instruction packs a message with a function name and its parameters to be remotely called, and places it in the outgoing-queue. The outgoingThread serializes the messages in the outgoing-queue, and send them over the network.

The receive instruction unpacks such message, and create a new operand-stack with the sequence's elements. Also, the interface of the sensor is placed at the top of the operand-stack. After this, a new runnable-process is created with the operand-stack as parameter, and the call instruction as the byte-code to run. The incomingThread is responsible to for the reception of data from the network, de-serialize it, and place it as a message in the incoming-queue.

```
case Opcodes.SEND:
        String sendFunctionName = (String) runningProcess.pop();
        int sendNumberParameters = ((Integer) runningProcess
                                            .pop()).intValue();
        Vector sendMessage = new Vector();
        if (sendNumberParameters > 0) {
                for (int sendIndex = 0; sendIndex < sendNumberParameters; sendIndex++) {
                        sendMessage.addElement(runningProcess.pop());
                }
        }
        sendMessage.addElement(new Integer(sendNumberParameters));
        sendMessage.addElement(sendFunctionName);
        outgoingQueue.put(sendMessage);
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.RECEIVE:
        if (!incomingQueue.isEmpty()) {
                Vector messageReceived = (Vector) incomingQueue
                                            .get();
                Function receiveFunction = sensorInterface
                                            .getFunctionByName((String) messageReceived
                                            .lastElement());
                RunnableProcess receiveRunnableProcess = new RunnableProcess(
                                                    receiveFunction);

                Enumeration messageElements = messageReceived
                                            .elements();
                while (messageElements.hasMoreElements()) {
                      receiveRunnableProcess.push(messageElements
                                                .nextElement());
                }
                receiveRunnableProcess.push(sensorInterface);
                runQueue.put(receiveRunnableProcess);
                break;
        }
```

```
        runningProcess.setProgramCounter(1);
        break;
```

This process of message serialize and de-serialize is provided by the class Serializer . Its implementation can be seen above, in which we have general methods to serialize and de-serialize a message, and more specific methods for the module structure.

```
protected class Serializer {

  protected static byte[] serialize(Vector message){ ...}

  protected static Vector deSerialize(byte[] serializedMessage){ ... }

  protected static byte[] serializeModule(Module module) throws IOException { ... }

  protected static Module deSerializeModule(DataInputStream dataInputStream){ ... }
}
```

The need to implement this class derives from the motivation to use our own framework independent format, namely, the Callas byte-code format itself.

4.5 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 [27]. 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 present a small example session that builds a Callas applications and runs it on the SunSPOT® simulation tool Solarium. We will use the Ping program (see Section 3.3), 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.

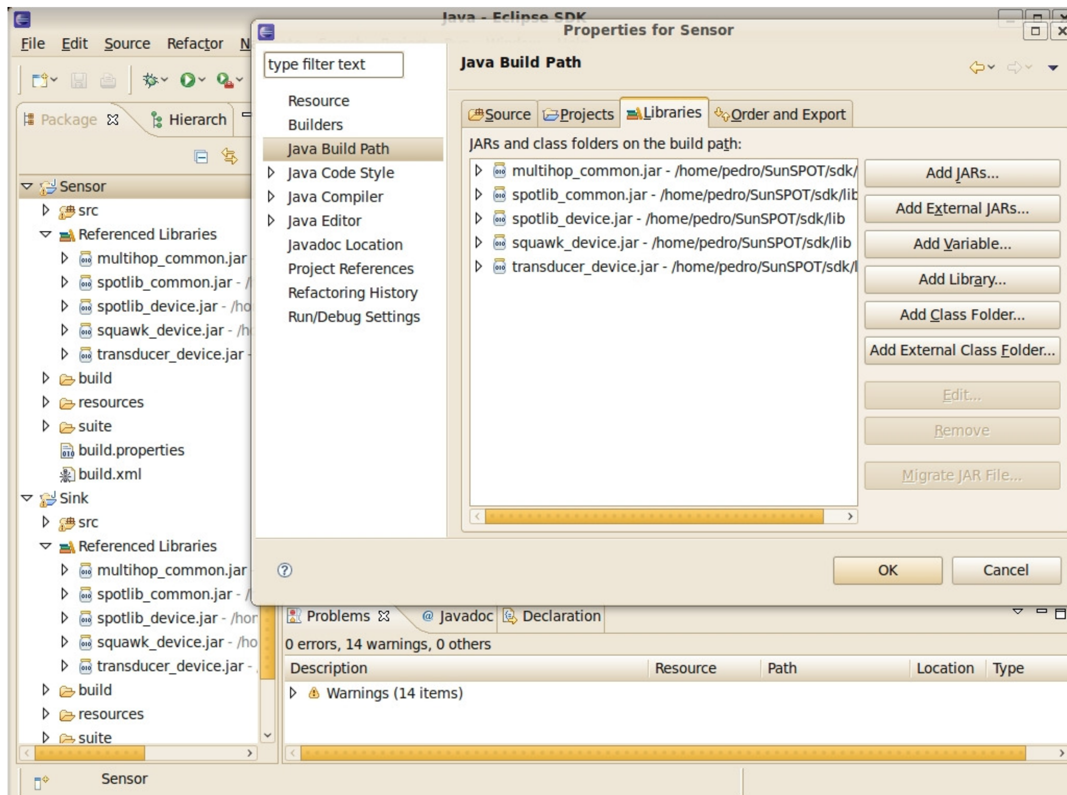


Figure 4.5: Referenced libraries.

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

```
MIDlet-Name: VirtualMachine
MIDlet-Version: 1.0.0
MIDlet-Vendor: Sun Microsystems Inc
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.

As we said, these projects have been tested in Solarium, provided by the SunSPOT®

SDK. Solarium provides an interface to remotely manage, or even emulate virtual SunSPOT® devices. It is also able to discover and display SunSPOTs® connected via USB and also via radio communication. Further, Solarium provides functionalities to load/unload software onto SunSPOTs®.

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 sensorboard, configurable leds, and communication via radio.

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

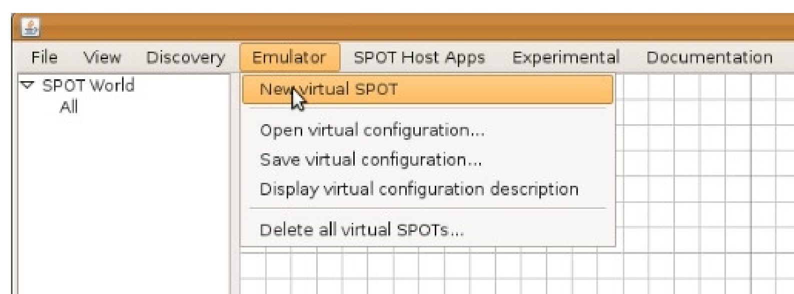


Figure 4.6: Creating a virtual SunSPOT.

A new virtual SPOT appears (Figure 4.7).

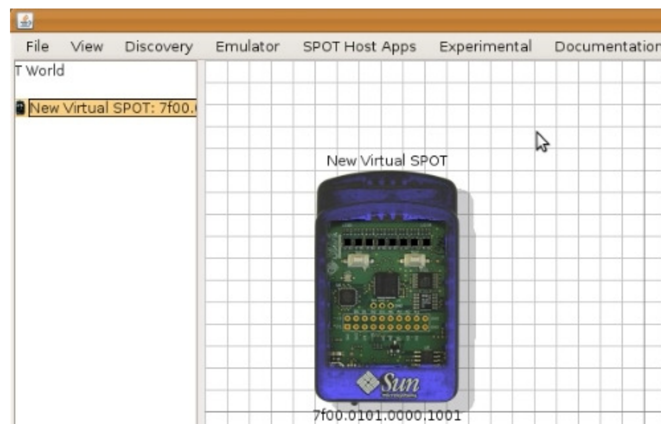


Figure 4.7: A new virtual SunSPOT.

We can manipulate this virtual SPOT as we like, using the mouse to move it anywhere in the displayed grid. Possible actions that can be done in a virtual SPOT are displayed in the right-click menu (Figure 4.8).

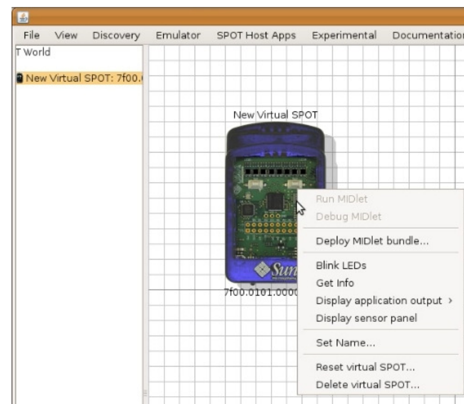
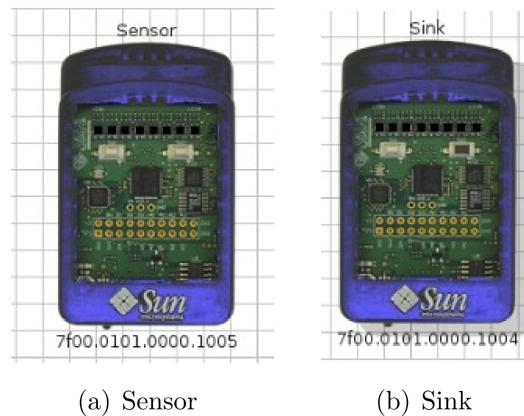


Figure 4.8: Possible actions.

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. The *Run MIDlet* action shows all the possible MIDlets of the project previously deployed, and allows to us to choose which of them we want to run.

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 4.9: 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.

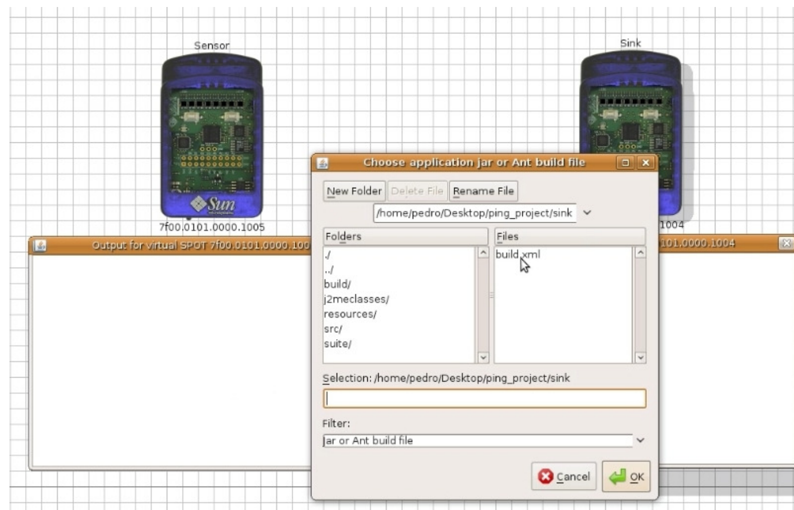


Figure 4.10: Deploy Sink project.

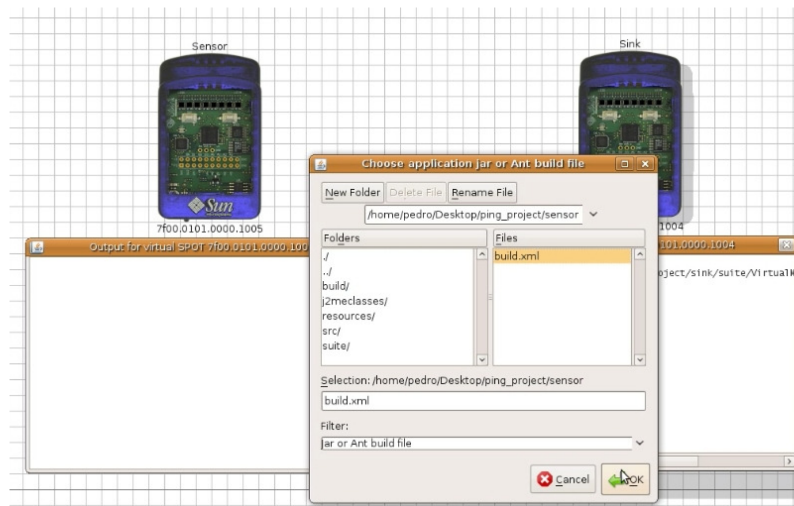


Figure 4.11: Deploy Sensor project.

To execute the Ping program, we hit the *RunMIDlet* command. Figure 4.12 shows the output from running the Ping program in the virtual SPOTs.

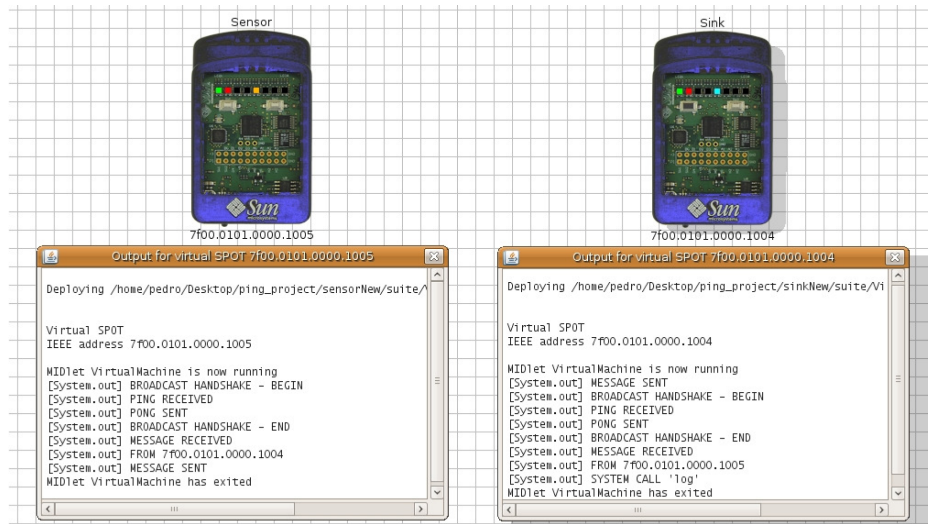


Figure 4.12: Result of Ping program execution.

This program was also tested in real SunSPOT® devices with the same output. A functional first version of the virtual machine was achieved. Further debugging, and more examples are necessary.

Chapter 5

Conclusions

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

We work 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 develop an implementation of a virtual machine for the SunSPOT® platform, which runs on top of a Squawk virtual machine.

We write some programs in Callas language and create correspondent projects, and test them in the SunSPOT® devices, and in the Solarium tool. All of the programs written are *type-safe*.

For future work, we intend 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.

Appendix A

The Callas Virtual Machine.

The Callas Virtual Machine is implemented inside of the package org.callas.vm, and it has as its constituents the following files:

- Program.java
- Module.java
- Function.java
- RunnableProcess.java
- RunningProcess.java
- TimedCall.java
- Opcodes.java
- Tags.java
- Serializer.java
- VirtualMachine.java

Next, we present this files.

```
package org.callas.vm;


import java.util.Vector;

/**
 * This class represents the structure of a Callas program.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date May 20, 2009
 *
 */
class Program {

        private int magicNumber;
        private byte version;
        private Vector types;
        private Vector modules;
        private byte[] byteCode;

        /**
         * The Constructor.
         * @param magicNumber
         * @param version
         * @param types
         * @param modules
         * @param byteCode
         */
        Program(int magicNumber, byte version, Vector types, Vector modules, byte[] byteCode) {
                this.magicNumber = magicNumber;
                this.version = version;
                this.types = types;
                this.modules = modules;
                this.byteCode = byteCode;

        }

        /**
         * Returns the program's magic number.
         * @return magicNumber
         */
         int getMagicNumber() {
                return this.magicNumber;
        }
        /**
         * Returns the program's version.
         * @return version
         */
         byte getVersion() {
                return this.version;
        }
        /**
         * Returns the program's types.
         * @return types
         */
         Vector getTypes() {
                return this.types;
        }

        /**
         * Returns the program's modules.
         * @return modules
         */
         Vector getModules() {
                return this.modules;
        }
        /**
         * Returns the program's byte-code.
         * @return byteCode
         */
         byte[] getByteCode(){
                return this.byteCode;
        }

        /**
         * Returns the program's first module.
```

```
        * @return module
        *
        */
        Module getFirstModule(){
                return (Module) this.modules.firstElement();
        }

        Module getModuleAt(int index){
                return (Module) this.modules.elementAt(index);

        }
}
```

```
package org.callas.vm;

import java.util.Enumeration;
import java.util.Vector;

/**
 * This class represents the structure of a Module object.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date May 20, 2009
 *
 */
class Module {

        private byte open;
        private Vector functions;

        /**
         * The Constructor.
         *
         */
        Module() {
                this.functions = new Vector();
        }

        /**
         * Define the module's functions.
         *
         * @param functions
         */
        void setFunctions(Vector functions) {
                this.functions = functions;
        }

        /**
         * Return the module's functions.
         *
         * @return functions
         */
        Vector getFunctions() {
                return this.functions;
        }

        /**
         * @param open
         *             the open to set
         */
        void setOpen(byte open) {
                this.open = open;
        }

        /**
         * @return the open
         */
        byte isOpen() {
                return open;
        }

        /**
         * Return the module's first function.
         *
         * @return function
         */

        Function getFirstFunction() {
                return (Function) this.functions.firstElement();
        }

        Function getFunctionAt(int index) {
                return (Function) this.functions.elementAt(index);
        }

        void setFunctionAt(Function function, int index) {
                this.functions.insertElementAt(function, index);
        }
```

```
        int getNumberOfFunctions() {
                return this.functions.size();
        }

        Function getFunctionByName(String functionName){
                Function function;
                Vector functions = this.getFunctions();
                Enumeration functionsElements = functions.elements();
                while (functionsElements.hasMoreElements()) {
                        function = (Function) functionsElements.nextElement();
                        if (function.getName().equals(functionName)) {
                                return function;
                        }
                }
                return null;
        }

}
```

```
package org.callas.vm;


import java.util.Vector;

/**
 * This class represents the structure of a Function.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date Mar 18, 2009
 */
class Function {

        private String name;
        private byte[] byteCode;
        private Vector symbols;
        private byte numberOfLocals;


        /**
         * The empty constructor.
         */
        Function(){
                // empty constructor
        }

        /**
         * The constructor.
         *
         * @param name
         * @param byteCode
         * @param symbols
         * @param numberOfLocals
         */
        Function(String name, byte[] byteCode, Vector symbols,
                        byte numberOfLocals) {
                this.name = name;
                this.byteCode = byteCode;
                this.symbols = symbols;
                this.numberOfLocals = numberOfLocals;
        }

        /**
         * Returns the function name.
         *
         * @return name
         */
        String getName() {
                return this.name;
        }

        /**
         * Returns the function's byte code.
         *
         * @return byteCode
         */
        byte[] getByteCode() {
                return this.byteCode;
        }

        /**
         * Returns the function's symbols.
         *
         * @return symbols
         */

        Vector getSymbols() {
                return this.symbols;
        }

        /**
         * Returns the number of locals of this function.
         *
         * @return numberOfLocals
         */
        byte getNumberOfLocals() {
```

```
		return this.numberOfLocals;
	}

	/**
	 * Change the function's name.
	 *
	 * @param name
	 */
	void setName(String name) {
		this.name = name;
	}

	/**
	 * Change the function's byte-code.
	 *
	 * @param byteCode
	 */
	void setByteCode(byte[] byteCode) {
		this.byteCode = byteCode;
	}

	/**
	 * Change the function's symbols.
	 *
	 * @param symbols
	 */
	void setSymbols(Vector symbols) {
		this.symbols = symbols;
	}

	/**
	 * Change the function's number of locals.
	 *
	 * @param numberOfLocals
	 */
	void setNumberOfLocals(byte numberOfLocals) {
		this.numberOfLocals = numberOfLocals;

	}

	void addSymbol(Object symbol){
		this.symbols.addElement(symbol);
	}

}
```

```
package org.callas.vm;

import java.util.Stack;
import java.util.Vector;

/**
 * This class represents the structure of a runnable process.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date May 30, 2009
 *
 */

class RunnableProcess {

        private Vector symbols;
        private byte[] byteCode;
        private Stack operandStack;
        private int numberOfLocals;

        /**
         * The constructor. Creates a process for the function to be executed.
         *
         * @param function
         */

        RunnableProcess(Function function) {
                this.symbols = function.getSymbols();
                this.byteCode = function.getByteCode();
                this.operandStack = new Stack();
                this.numberOfLocals = function.getNumberOfLocals();
        }

        /**
         *
         * @return symbols
         */
        Vector getSymbols() {
                return this.symbols;
        }

        /**
         *
         * @return size
         */
        int getEnvironmentSize() {
                return this.numberOfLocals + this.operandStack.size();
        }

        /**
         *
         * @return bytecode
         */
        byte[] getByteCode() {
                return this.byteCode;
        }

        /**
         *
         * @param symbols
         */
        void setSymbols(Vector symbols) {
                this.symbols = symbols;
        }

        /**
         *
         * @param byteCode
         */
        void setByteCode(byte[] byteCode) {
                this.byteCode = byteCode;
        }

        /**
         * @param operandStack
         *            the operandStack to set
```

```
	 */
	void setOperandStack(Stack operandStack) {
		this.operandStack = operandStack;
	}

	/**
	 * @return the operandStack
	 */
	Stack getOperandStack() {
		return operandStack;
	}

	void setNumberOfLocals(int numberOfLocals) {
		this.numberOfLocals = numberOfLocals;
	}

	int getNumberOfLocals() {
		return numberOfLocals;
	}

	Object pop() {
		return this.operandStack.pop();
	}

	void push(Object value) {
		this.operandStack.push(value);
	}
}
```

```
package org.callas.vm;


import java.util.Stack;
import java.util.Vector;

/**
 * This class represents the structure of a running process.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date May 30, 2009
 *
 */

class RunningProcess {

        private Vector symbols;
        private Vector environment;
        private Stack operandStack;
        private int programCounter;
        private byte[] byteCode;

        /**
         * The constructor.
         *
         * @param runnableProcess
         * @param parameters
         */

        RunningProcess(RunnableProcess runnableProcess) {
                this.symbols = runnableProcess.getSymbols();
                this.byteCode = runnableProcess.getByteCode();
                this.environment = new Vector(runnableProcess.getEnvironmentSize());
                this.environment.setSize(runnableProcess.getEnvironmentSize());
                this.programCounter = 0;
                this.operandStack   = runnableProcess.getOperandStack();
        }


        Object pop(){
                return this.operandStack.pop();
        }

        void push(Object value){
                this.operandStack.push(value);
        }

        boolean stackEmpty(){
                return this.operandStack.empty();
        }
        /**
         *
         * @return byte
         */
        byte getByteAt(int index){
                return this.byteCode[index];
        }

        /**
         * @param symbols
         *            the symbols to set
         */
         void setSymbols(Vector symbols) {
                this.symbols = symbols;
        }

        /**
         * @return the symbols
         */
         Vector getSymbols() {
                return symbols;
        }

        Object getSymbolAt(int index){
                return this.symbols.elementAt(index);
        }
```

```
        Object getEnvironmentAt(int index){
                return this.environment.elementAt(index);
        }

        void setEnvironmentAt(Object obj,int index){
                this.environment.insertElementAt(obj, index);
        }

        /**
         * @param operandStack
         */
         void setOperandStack(Stack operandStack) {
                this.operandStack = operandStack;
        }

        /**
         * @return the operandStack
         */
         Stack getOperandStack() {
                return operandStack;
        }

        /**
         * @param programCounter
         */
         void setProgramCounter(int programCounter) {
                this.programCounter += programCounter;
        }

        /**
         * @return the programCounter
         */
         int getProgramCounter() {
                return programCounter;
        }

        /**
         * @param byteCode
         */
         void setByteCode(byte[] byteCode) {
                this.byteCode = byteCode;
        }

        /**
         * @return the byteCode
         */
         byte[] getByteCode() {
                return byteCode;
        }

         void setEnvironment(Vector environment) {
                this.environment = environment;
        }

         Vector getEnvironment() {
                return environment;
        }

}
```

```
package org.callas.vm;


import java.util.Vector;

/**
 * This class represents an event object, this event is called periodically.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date Mar 4, 2009
 *
 */

class TimedCall extends Thread {

        private Module sensorInterface;
        private String functionName;
        private Vector parameters;
        private int period;
        private int expire;
        private static final int ONE_SECOND = 1000;

        /**
         * The Constructor. Creates a thread to each event that is created.
         *
         * @param message
         * @param period
         * @param expire
         */
        TimedCall(Module sensorInterface, String functionName, Vector parameters,
                        int period, int expire) {
                this.sensorInterface = sensorInterface;
                this.functionName = functionName;
                this.parameters = parameters;
                this.period = period;
                this.expire = expire;
                // launches thread
                //this.run();
        }

         public void run() {

                VirtualMachine vm = VirtualMachine.getVirtualMachine();
                int i=1;
                while (true) {

                        try {
                                this.sleep(this.period*TimedCall.ONE_SECOND);
                        } catch (InterruptedException e) {}

                        if(this.expire <= (this.period*i)){
                                break;
                        }
                        i++;
                        // put timed call in the queue of events to be processed
                        RunnableProcess newCall= new RunnableProcess (vm.searchFunctionInModule(sensorInterface, functionName));
                        int parSize = parameters.size();
                        for(int j=parSize;j<parSize;j--){
                                newCall.getOperandStack().push(parameters.elementAt(j));
                        }

                        vm.getRunQueue().put(newCall);

                }
                this.yield();
                this.interrupt();

        }

        /**
         * Returns the TimedCall's function name.
         *
         * @return functionName
         */
        protected String getFunctionName() {
                return this.functionName;
```

```
        }
}
```

```
package org.callas.vm;


/**
 * The interface to all the opcodes.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date Mar 18, 2009
 *
 */

 interface Opcodes {
        static final byte IADD = 0x00;
        static final byte FADD = 0x01;
        static final byte ISUB = 0x02;
        static final byte FSUB = 0x03;
        static final byte IMUL = 0x04;
        static final byte FMUL = 0x05;
        static final byte IDIV = 0x06;
        static final byte FDIV = 0x07;
        static final byte IMOD = 0x08;
        static final byte FMOD = 0x09;
        static final byte INOT = 0x10;
        static final byte BNOT = 0x11;
        static final byte IAND = 0x12;
        static final byte BAND = 0x13;
        static final byte IOR = 0x14;
        static final byte BOR = 0x15;
        static final byte IXOR = 0x16;
        static final byte BXOR = 0x17;
        static final byte POP = 0x18;
        static final byte DUP = 0x19;
        static final byte SWAP = 0x20;
        static final byte LOAD = 0x21;
        static final byte HALT = 0x22;
        static final byte LOADC = 0x23;
        static final byte CALL = 0x24;
        static final byte STORE = 0x25;
        static final byte SYSTEM = 0x26;
        static final byte TIMER = 0x27;
        static final byte IFTRUE = 0x28;
        static final byte EQ = 0x29;
        static final byte NEQ = 0x30;
        static final byte LT = 0x31;
        static final byte GT = 0x32;
        static final byte LEQ = 0x33;
        static final byte GEQ = 0x34;
        static final byte GOTO = 0x35;
        static final byte INSTALL = 0x36;
        static final byte RECEIVE = 0x37;
        static final byte RETURN = 0x38;
        static final byte SEND = 0x39;
        static final byte LOADM = 0x40;
        static final byte LOADB = 0x41;
        static final byte STOREB = 0x42;

}
```

```java
package org.callas.vm;


/**
 * The interface to all the tags that occurs in the byte-code.
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date May 24, 2009
 *
 */
interface Tags {

        static final byte MAGIC = 0x00;
        static final byte BOOL = 0x01;
        static final byte INTEGER = 0x02;
        static final byte STRING = 0x03;
        static final byte FLOAT = 0x04;
        static final byte MODULE = 0x05;
        static final byte VERSION = 0x06;
        static final byte SYMBOLS = 0x07;
        static final byte LOCALS = 0x08;
        static final byte TYPES = 0x09;
        static final byte CODE = 0x10;
        static final byte FUNCTION = 0x11;
        static final byte MODULES = 0x12;
        static final byte OPEN = 0x13;
}
```

```java
package org.callas.vm;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Vector;

import org.callas.vm.exceptions.MalformedByteCodeException;



/**
 * This class represents the serializer to the messages exchanged between
 * sensors.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date Mar 18, 2009
 *
 */

class Serializer {

        /**
         * Given a message, this method serializes it into a byte array
         *
         * @param message
         * @return byte[]
         * @throws Exception
         * @throws MalformedByteCodeException
         */

        protected static byte[] serialize(Vector message)
                        throws MalformedByteCodeException, Exception {

                // streams used to write the byte array
                ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
                DataOutputStream dataOutStream = new DataOutputStream(byteOutStream);
                VirtualMachine vm = VirtualMachine.getVirtualMachine();
                // magic number
                dataOutStream.writeByte(Tags.MAGIC);
                dataOutStream.writeInt(vm.getMagicNumber());
                int numberOfParameters = message.size();

                // number of parameters
                dataOutStream.writeByte(Tags.INTEGER);
                dataOutStream.writeInt(numberOfParameters);

                // parameters
                for (int i = 0; i < numberOfParameters; i++) {
                        Object o = message.elementAt(i);
                        if (o instanceof String) {
                                // if the element is a string
                                dataOutStream.writeByte(Tags.STRING);
                                dataOutStream.writeUTF(((String) o).toString());
                        } else if (o instanceof Integer) {
                                // if the element is an integer
                                dataOutStream.writeByte(Tags.INTEGER);
                                dataOutStream.writeInt(((Integer) o).intValue());
                        } else if (o instanceof Float) {
                                // if the element is a float
                                dataOutStream.writeByte(Tags.FLOAT);
                                dataOutStream.writeFloat(((Float) o).floatValue());
                        } else if (o instanceof Boolean) {
                                // if the element is a boolean
                                dataOutStream.writeByte(Tags.BOOL);
                                dataOutStream.writeBoolean(((Boolean) o).booleanValue());
                        } else if (o instanceof Module) {
                                // if the element is an module
                                Module module = (Module) o;
                                dataOutStream.write(serializeModule(module));
                        } else {
                                throw new IOException("Cannot serialize " + "object of type "
                                                + o.getClass().getName());
```

```
                }

        }
        dataOutStream.flush();
        dataOutStream.close();
        return byteOutStream.toByteArray();
}

/**
 * Given a byte array, this method de-serializes it into a message.
 *
 * @param serializedMesage
 * @return message
 * @throws Exception
 * @throws MalformedByteCodeException
 */

protected static Vector deSerialize(byte[] serializedMessage)
                throws MalformedByteCodeException, Exception {

        boolean differentSex;
        Vector message = null;

        ByteArrayInputStream byteInputStream = new ByteArrayInputStream(
                        serializedMessage);
        DataInputStream dataInputStream = new DataInputStream(byteInputStream);
        VirtualMachine vm = VirtualMachine.getVirtualMachine();
        // magic number
        if (dataInputStream.readByte() == Tags.MAGIC) {
                long magicNumber = dataInputStream.readInt();
                if (magicNumber != vm.getMagicNumber())
                        differentSex = true;
                // TODO swap byte order
        }

        // message's parameters
        if (dataInputStream.readByte() == Tags.INTEGER) {

                message = new Vector();
                // number of parameters
                int numberOfParameters = dataInputStream.readInt();
                for (int i = 0; i < numberOfParameters; i++) {
                        int type = dataInputStream.readByte();
                        if (type == Tags.INTEGER) {
                                // if element is an integer
                            message.addElement(new Integer(dataInputStream.readInt()));
                        } else if (type == Tags.FLOAT) {
                                // if element is a float
                            message.addElement(new Float(dataInputStream.readFloat()));
                        } else if (type == Tags.STRING) {
                                // if element is a string
                            message.addElement((String)dataInputStream.readUTF());
                        } else if (type == Tags.BOOL) {
                                // if element is a string
                            message.addElement(new Boolean(dataInputStream.readBoolean()));
                        }
                        else if (type == Tags.MODULE) {
                                // if element is an module
                            message.addElement((Object)deSerializeModule(dataInputStream));
                        }
                }
        }
        dataInputStream.close();

        return message;

}

/**
 * Given the code, this method serializes it into a byte array.
 *
 * @param module
 * @return byte[]
 * @throws IOException
 */

protected static byte[] serializeModule(Module module) throws IOException {
```

```
        // streams used to write the byte array
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        DataOutputStream dataOutStream = new DataOutputStream(byteOutStream);
        // number of functions in module
        dataOutStream.writeByte(Tags.MODULE);
        dataOutStream.writeInt(module.getFunctions().size());

        // functions
        Function function;
        Enumeration functionsElements = module.getFunctions().elements();

        while (functionsElements.hasMoreElements()) {
                function = (Function) functionsElements.nextElement();
                dataOutStream.writeByte(Tags.FUNCTION);
                dataOutStream.writeByte(Tags.STRING);
                dataOutStream.writeUTF(function.getName());
                dataOutStream.writeByte(Tags.LOCALS);
                dataOutStream.writeByte(function.getNumberOfLocals());
                dataOutStream.writeByte(Tags.CODE);
                dataOutStream.writeInt(function.getByteCode().length);
                dataOutStream.write(function.getByteCode());
                dataOutStream.writeByte(Tags.SYMBOLS);
                dataOutStream.writeByte(function.getSymbols().size());
                for (int i = 0; i < function.getSymbols().size(); i++) {
                        Object o = function.getSymbols().elementAt(i);
                        if (o instanceof String) {
                                // the element is a string
                                dataOutStream.writeByte(Tags.STRING);
                                dataOutStream.writeUTF((String) o);
                        } else if (o instanceof Integer) {
                                // the element is an integer
                                dataOutStream.writeByte(Tags.INTEGER);
                                dataOutStream.writeInt(((Integer) o).intValue());
                        } else if (o instanceof Float) {
                                // the element is a float
                                dataOutStream.writeByte(Tags.FLOAT);
                                dataOutStream.writeFloat(((Float) o).floatValue());
                        } else if (o instanceof Boolean) {
                                // the element is a boolean
                                dataOutStream.writeByte(Tags.BOOL);
                                dataOutStream.writeBoolean(((Boolean) o).booleanValue());
                        } else {
                                throw new IOException("Cannot serialize "
                                                + "object of type " + o.getClass().getName());
                        }
                        i += 1;
                }
        }
        dataOutStream.flush();
        dataOutStream.close();
        // return byte array with module serialized
        return byteOutStream.toByteArray();
}

/**
 * Given a DataInputStream, this method de-serialize's it into a module.
 *
 * @param dataInputStream
 * @return code
 * @throws IOException
 */

protected static Module deSerializeModule(DataInputStream dataInputStream)
                throws IOException {

        String functionName = null;
        byte numberOfLocals = 0;
        int byteCodeSize;
        byte[] byteCode = null;
        byte numberOfSymbols;
        Vector symbols = null;
        Vector functions = new Vector();
        if (dataInputStream.readByte() == Tags.MODULE) {
            int numberOfFunctions = dataInputStream.readInt();
            for (int i = 0; i < numberOfFunctions; i++) {
                if (dataInputStream.readByte() == Tags.FUNCTION) {
```

```
                    if (dataInputStream.readByte() == Tags.STRING) {
                        functionName = dataInputStream.readUTF();
                        if (dataInputStream.readByte() == Tags.LOCALS) {
                            numberOfLocals = dataInputStream.readByte();
                            if (dataInputStream.readByte() == Tags.CODE) {
                                byteCodeSize = dataInputStream.readInt();
                                dataInputStream.read(byteCode, 0, byteCodeSize);
                                if (dataInputStream.readByte() == Tags.SYMBOLS) {
                                    numberOfSymbols = dataInputStream
                                        .readByte();
                                    symbols = new Vector();
                                    for (int j = 0; j < numberOfSymbols; j++) {
                                        switch (dataInputStream.readByte()) {
                                        case Tags.STRING:
                                            symbols.addElement(dataInputStream
                                                                .readUTF());
                                            break;

                                        case Tags.INTEGER:
                                            symbols.addElement(new Integer(dataInputStream
                                                                            .readInt()));
                                            break;

                                        case Tags.FLOAT:
                                            symbols.addElement(new Float(dataInputStream
                                                                          .readFloat()));
                                            break;

                                        case Tags.BOOL:
                                            symbols.addElement(new Boolean(dataInputStream
                                                                            .readBoolean()));
                                            break;

                                        default:
                                            // TODO should not enter here —¿
                                            // add exception
                                            break;
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
                functions.addElement(new Function(functionName, byteCode,
                                                  symbols, numberOfLocals));
                symbols = null;
            }
        }

        Module module = new Module();
        module.setFunctions(functions);
        return module;
    }
}
```

```
package org.callas.vm;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Stack;
import java.util.Vector;

import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import javax.microedition.io.StreamConnection;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import org.callas.vm.exceptions.MalformedByteCodeException;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import com.sun.spot.sensorboard.peripheral.LEDColor;
import com.sun.spot.util.Queue;
import com.sun.spot.util.Utils;

/**
 * This class implements the Callas Virtual Machine.
 *
 * @author Pedro Gomes (prg@dcc.fc.up.pt)
 * @date Mar 18, 2009
 *
 */

public class VirtualMachine extends MIDlet {

        private Thread mainLoopThread;
        private Thread incomingThread;
        private Thread outgoingThread;

        // variable declaration
        private Queue incomingQueue;
        private Queue outgoingQueue;
        private Queue runQueue;
        private Stack callStack;
        private Vector timers;
        private Module sensorInterface;
        private Program program;
        private int magicNumber;

        private static VirtualMachine vm = null;

        public static VirtualMachine getVirtualMachine() {
                return vm;
        }

        public void execute(byte[] byteCode) {
                // extract magic number, version, types
                program = extractProgramInformation(byteCode);
                mainLoopThread.start();
                incomingThread.start();
                outgoingThread.start();
        }

        /**
         * The constructor creates all the objects needed to run a Callas Virtual
         * Machine.
         *
         * @throws MalformedByteCodeException
         * @throws FileNotFoundException
         */

        public VirtualMachine() throws MalformedByteCodeException {
                // FileNotFoundException

                // STRUCTURES INITIALIZATION
                runQueue = new Queue();
```

```
incomingQueue = new Queue();
outgoingQueue = new Queue();
callStack = new Stack();
sensorInterface = new Module();
timers = new Vector();
/**
 * The main thread, it is responsible to process all the functions of a
 * program.
 */
mainLoopThread = new Thread(new Runnable() {

        /**
         * Runs the given byte code.
         *
         */

        public void run() {

                // local variable initialization
                RunnableProcess runnableProcess;
                RunningProcess runningProcess;
                Stack operandStack;
                int programCounter;
                Vector frame;
                // LEDS
                ITriColorLED[] leds = EDemoBoard.getInstance().getLEDs();

                Module firstModule = program.getFirstModule();

                Function firstFunction = firstModule.getFirstFunction();

                runningProcess = new RunningProcess(new RunnableProcess(
                                firstFunction));
                callStack.push(runningProcess);

                // here we read the byte code
                while (true) {

                        // get a new process to execute
                        if (runningProcess == null) {

                                if (callStack.empty()) {

                                        while (runQueue.isEmpty())
                                                ;
                                        runnableProcess = (RunnableProcess) runQueue.get();
                                        runningProcess = new RunningProcess(runnableProcess);
                                        callStack.addElement(runningProcess);
                                        break;

                                } else {
                                        runningProcess = (RunningProcess) callStack
                                                        .firstElement();
                                }
                        }
                        // the main switch to execute all the byte code
                        switch (runningProcess.getByteAt(runningProcess
                                        .getProgramCounter())) {

                        case Opcodes.IADD:
                                int iaddOperand1 = ((Integer) runningProcess.pop())
                                                .intValue();
                                int iaddOperand2 = ((Integer) runningProcess.pop())
                                                .intValue();
                                runningProcess.push(new Integer(iaddOperand1
                                                + iaddOperand2));
                                runningProcess.setProgramCounter(1);

                                break;

                        case Opcodes.FADD:
                                float faddOperand1 = ((Float) runningProcess.pop())
                                                .floatValue();
                                float faddOperand2 = ((Float) runningProcess.pop())
                                                .floatValue();
                                runningProcess.push(new Float(faddOperand1
                                                + faddOperand2));
```

```
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.ISUB:
        int isubOperand1 = ((Integer) runningProcess.pop())
                        .intValue();
        int isubOperand2 = ((Integer) runningProcess.pop())
                        .intValue();
        runningProcess.push(new Integer(isubOperand1
                        - isubOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.FSUB:
        float fsubOperand1 = ((Float) runningProcess.pop())
                        .floatValue();
        float fsubOperand2 = ((Float) runningProcess.pop())
                        .floatValue();
        runningProcess.push(new Float(fsubOperand1
                        - fsubOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.IMUL:
        int imulOperand1 = ((Integer) runningProcess.pop())
                        .intValue();
        int imulOperand2 = ((Integer) runningProcess.pop())
                        .intValue();
        runningProcess.push(new Integer(imulOperand1
                        * imulOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.FMUL:
        float fmulOperand1 = ((Float) runningProcess.pop())
                        .floatValue();
        float fmulOperand2 = ((Float) runningProcess.pop())
                        .floatValue();
        runningProcess.push(new Float(fmulOperand1
                        * fmulOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.IDIV:
        int idivOperand1 = ((Integer) runningProcess.pop())
                        .intValue();
        int idivOperand2 = ((Integer) runningProcess.pop())
                        .intValue();
        runningProcess.push(new Integer(idivOperand1
                        / idivOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.FDIV:
        float fdivOperand1 = ((Float) runningProcess.pop())
                        .floatValue();
        float fdivOperand2 = ((Float) runningProcess.pop())
                        .floatValue();
        runningProcess.push(new Float(fdivOperand1
                        / fdivOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.IMOD:
        int imodOperand1 = ((Integer) runningProcess.pop())
                        .intValue();
        int imodOperand2 = ((Integer) runningProcess.pop())
                        .intValue();
        runningProcess.push(new Integer(imodOperand1
                        % imodOperand2));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.FMOD:
        float fmodOperand1 = ((Float) runningProcess.pop())
                        .floatValue();
        float fmodOperand2 = ((Float) runningProcess.pop())
```

```
                    .floatValue();
          runningProcess.push(new Float(fmodOperand1
                    % fmodOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.INOT:
          break;

case Opcodes.BNOT:
          runningProcess.push(new Boolean(
                    !((Boolean) runningProcess.pop())
                                        .booleanValue()));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.IAND:
          int iandOperand1 = ((Integer) runningProcess.pop())
                    .intValue();
          int iandOperand2 = ((Integer) runningProcess.pop())
                    .intValue();
          runningProcess.push(new Integer(iandOperand1
                    & iandOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.BAND:
          boolean bandOperand1 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          boolean bandOperand2 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          runningProcess.push(new Boolean(bandOperand1
                    & bandOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.IOR:
          int iorOperand1 = ((Integer) runningProcess.pop())
                    .intValue();
          int iorOperand2 = ((Integer) runningProcess.pop())
                    .intValue();
          runningProcess.push(new Integer(iorOperand1
                    | iorOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.BOR:
          boolean borOperand1 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          boolean borOperand2 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          runningProcess.push(new Boolean(borOperand1
                    | borOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.IXOR:
          int ixorOperand1 = ((Integer) runningProcess.pop())
                    .intValue();
          int ixorOperand2 = ((Integer) runningProcess.pop())
                    .intValue();
          runningProcess.push(new Integer(ixorOperand1
                    ^ ixorOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.BXOR:
          boolean bxorOperand1 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          boolean bxorOperand2 = ((Boolean) runningProcess.pop())
                    .booleanValue();
          runningProcess.push(new Boolean(bxorOperand1
                    ^ bxorOperand2));
          runningProcess.setProgramCounter(1);
          break;

case Opcodes.POP:
```

```
        runningProcess.pop();
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.DUP:
        Object dupOperand = runningProcess.pop();
        runningProcess.push(dupOperand);
        runningProcess.push(dupOperand);
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.SWAP:
        Object swapOperand1 = runningProcess.pop();
        Object swapOperand2 = runningProcess.pop();
        runningProcess.push(swapOperand1);
        runningProcess.push(swapOperand2);
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.LOADC:
        byte loadcIndex = runningProcess
                        .getByteAt(runningProcess.getProgramCounter() + 1);
        runningProcess.push(runningProcess
                        .getSymbolAt(loadcIndex));
        runningProcess.setProgramCounter(2);
        break;

case Opcodes.LOAD:
        byte loadIndex = runningProcess
                        .getByteAt(runningProcess.getProgramCounter() + 1);
        runningProcess.push(runningProcess
                        .getEnvironmentAt(loadIndex));
        runningProcess.setProgramCounter(2);
        break;

case Opcodes.LOADM:
        Module moduleLoaded = (Module) program
                        .getModuleAt(runningProcess
                                        .getByteAt(runningProcess
                                                        .getProgramCounter() + 1));

        // needs to load free vars
        // number of functions of the module
        if (moduleLoaded.isOpen() == 1) {
                int loadmNumberFunctions = moduleLoaded
                                .getNumberOfFunctions();
                int loadmFreeVar;
                for (int loadmIndex = 0; loadmIndex <
                         loadmNumberFunctions; loadmIndex++) {
                        Function function = (Function) moduleLoaded
                                        .getFunctionAt(loadmIndex);
                        loadmFreeVar = ((Integer) runningProcess.pop())
                                        .intValue();
                        if (loadmFreeVar > 0) {
                                for (int loadmIndex2 = 0; loadmIndex2
                                         < loadmFreeVar; loadmIndex2++) {
                                        function
                                            .addSymbol(runningProcess.pop());
                                }
                                moduleLoaded.setFunctionAt(function,
                                                loadmIndex);
                        }

                }
        }
        runningProcess.push(moduleLoaded);
        runningProcess.setProgramCounter(2);

        break;

case Opcodes.STORE:
        byte storeIndex = runningProcess
                        .getByteAt(runningProcess.getProgramCounter() + 1);
        runningProcess.setEnvironmentAt(runningProcess.pop(),
                        storeIndex);
        runningProcess.setProgramCounter(2);
        break;
```

```
case Opcodes.EQ:
        Object ifeqValue1 = runningProcess.pop();
        Object ifeqValue2 = runningProcess.pop();
        if (ifeqValue1 == ifeqValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.NEQ:
        Object ifneqValue1 = runningProcess.pop();
        Object ifneqValue2 = runningProcess.pop();
        if (ifneqValue1 != ifneqValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.LT:
        int ifltValue1 = ((Integer) runningProcess.pop())
                        .intValue();
        int ifltValue2 = ((Integer) runningProcess.pop())
                        .intValue();
        if (ifltValue1 < ifltValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.GT:
        int ifgtValue1 = ((Integer) runningProcess.pop())
                        .intValue();
        int ifgtValue2 = ((Integer) runningProcess.pop())
                        .intValue();
        if (ifgtValue1 > ifgtValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.LEQ:
        int ifleqValue1 = ((Integer) runningProcess.pop())
                        .intValue();
        int ifleqValue2 = ((Integer) runningProcess.pop())
                        .intValue();
        if (ifleqValue1 <= ifleqValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.GEQ:
        int ifgeqValue1 = ((Integer) runningProcess.pop())
                        .intValue();
        int ifgeqValue2 = ((Integer) runningProcess.pop())
                        .intValue();
        if (ifgeqValue1 >= ifgeqValue2)
                runningProcess.push(new Boolean(true));
        else
                runningProcess.push(new Boolean(false));
        runningProcess.setProgramCounter(1);
        break;

case Opcodes.IFTRUE:
        boolean iftrueValue = ((Boolean) runningProcess.pop())
                        .booleanValue();
        int iftrueOffset = byteArrayToInt(new byte[] {
                        runningProcess.getByteAt(runningProcess
                                        .getProgramCounter() + 1),
                        runningProcess.getByteAt(runningProcess
                                        .getProgramCounter() + 2),
```

```
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 3),
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 4) }, 0);
        if (iftrueValue)
                runningProcess.setProgramCounter(iftrueOffset);
        else
                runningProcess.setProgramCounter(5);
        break;

case Opcodes.GOTO:
        // System.out.println("byteCode["
        // + runningProcess.getrunningProcess.programCounter()
        // + "] : goto");
        int gotoOffset = byteArrayToInt(new byte[] {
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 1),
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 2),
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 3),
                    runningProcess.getByteAt(runningProcess
                                    .getProgramCounter() + 4) }, 0);
        runningProcess.setProgramCounter(gotoOffset);
        break;

case Opcodes.INSTALL:
        Module originalModule = (Module) runningProcess.pop();

        Module toInstallModule = (Module) runningProcess.pop();

        Enumeration toInstallElements = toInstallModule
                    .getFunctions().elements();
        while (toInstallElements.hasMoreElements()) {
                Enumeration originalElements = originalModule
                            .getFunctions().elements();
                Function toInstallFunction = (Function) toInstallElements
                            .nextElement();
                while (originalElements.hasMoreElements()) {
                        Function originalFunction =
                            (Function) originalElements
                                        .nextElement();
                        if (toInstallFunction.getName().equals(
                                        originalFunction.getName())) {
                                int functionIndex = originalModule
                                        .getFunctions().indexOf(
                                        originalFunction);
                                originalModule.setFunctionAt(
                                        toInstallFunction, functionIndex);
                                toInstallFunction = null;
                                break;
                        }
                }
                if (toInstallFunction != null) {

                        originalModule.getFunctions().addElement(
                                        toInstallFunction);
                }

        }
        runningProcess.push(originalModule);
        runningProcess.setProgramCounter(1);

        break;

case Opcodes.SEND:
        String sendFunctionName = (String) runningProcess.pop();
        int sendNumberParameters = ((Integer) runningProcess
                    .pop()).intValue();
        Vector sendMessage = new Vector();
        if (sendNumberParameters > 0) {
                for (int sendIndex = 0; sendIndex
                         < sendNumberParameters; sendIndex++) {
                        sendMessage.addElement(runningProcess.pop());
                }
        }
        sendMessage
```

```
                    .addElement(new Integer(sendNumberParameters));
          sendMessage.addElement(sendFunctionName);

          outgoingQueue.put(sendMessage);

          runningProcess.setProgramCounter(1);
          // uncomment these lines if you want to see the leds
          // blinking during a send
          // leds[0].setColor(LEDColor.GREEN);
          // leds[0].setOn();
          break;

case Opcodes.SYSTEM:
          // to be done properly
          String systemCall = (String) runningProcess.pop();
          int systemNumberParameters = ((Integer) runningProcess
                    .pop()).intValue();
          if (systemNumberParameters > 0) {
                    Vector systemParameters = new Vector();
                    for (int systemIndex = 0; systemIndex <
                              systemNumberParameters; systemIndex++) {
                              systemParameters.addElement(runningProcess
                                        .pop());
                    }
          }

          // to be done properly
          if (systemCall.equals("log")) {
                    runningProcess.push("LOG");
                    System.out.println("SYSTEM CALL 'log'");
                    leds[4].setColor(LEDColor.CYAN);
                    leds[4].setOn();
          } else if (systemCall.equals("mac")) {
                    long macLongValue = Spot.getInstance()
                              .getRadioPolicyManager().getIEEEAddress();
                    String macStringValue = Long.toString(macLongValue);
                    runningProcess.push(macStringValue);
                    System.out.println("MAC : " + macLongValue);
                    leds[4].setColor(LEDColor.YELLOW);
                    leds[4].setOn();
          } else if (systemCall.equals("ping")) {
                    System.out.println("ping sent");

          }

          runningProcess.setProgramCounter(1);

          break;

case Opcodes.TIMER:
          int timerExpire = ((Integer) runningProcess.pop())
                    .intValue();
          int timerPeriod = ((Integer) runningProcess.pop())
                    .intValue();
          Module timerBrane = (Module) runningProcess.pop();
          String timerFunctionName = (String) runningProcess
                    .pop();
          int numberOfParameters = ((Integer) runningProcess
                    .pop()).intValue();
          // gets all the parameters for the function
          Vector timerParameters = new Vector();
          if (numberOfParameters > 0) {
                    for (int timerIndex = 0; timerIndex
                              < numberOfParameters; timerIndex++) {
                              timerParameters
                                        .addElement(runningProcess.pop());
                    }
          }
          runningProcess.setProgramCounter(1);
          // check if timers has a timed call for the same
          // function
          if (timers.isEmpty()) {
                    TimedCall timedCall = new TimedCall(timerBrane,
                              timerFunctionName, timerParameters,
                              timerPeriod, timerExpire);
                    timers.addElement(timedCall);
                    timedCall.start();
```

```
                        // System.out.println("SIZE OF TIMERS: "
                        // + timers.size());

                } else {
                        Enumeration timersElements = timers.elements();
                        TimedCall element;
                        while (timersElements.hasMoreElements()) {
                                element = (TimedCall) timersElements
                                                .nextElement();
                                if (timerFunctionName.equals(element
                                                .getFunctionName())) {
                                        // CREATE EXCEPTION
                                }
                        }
                        timers.addElement(new TimedCall(timerBrane,
                                        timerFunctionName, timerParameters,
                                        timerPeriod, timerExpire));

                }
                break;

        case Opcodes.LOADB:
                runningProcess.push(sensorInterface);
                runningProcess.setProgramCounter(1);
                break;

        case Opcodes.STOREB:
                sensorInterface = (Module) runningProcess.pop();
                runningProcess.setProgramCounter(1);
                break;

        case Opcodes.RECEIVE:
                if (!incomingQueue.isEmpty()) {

                        Vector messageReceived = (Vector) incomingQueue
                                        .get();
                        // TODO getFunctionByName
                        Function receiveFunction = sensorInterface
                                        .getFunctionByName((String) messageReceived
                                        .lastElement());

                        RunnableProcess receiveRunnableProcess = new RunnableProcess(
                                        receiveFunction);

                        Enumeration messageElements = messageReceived
                                        .elements();
                        while (messageElements.hasMoreElements()) {
                                receiveRunnableProcess.push(messageElements
                                                .nextElement());
                        }
                        receiveRunnableProcess.push(sensorInterface);

                        runQueue.put(receiveRunnableProcess);

                        break;
                }

                runningProcess.setProgramCounter(1);

                // uncomment these lines if you want to see the leds
                // blinking during receive
                // leds[1].setColor(LEDColor.RED);
                // leds[1].setOn();
                // Utils.sleep(100);
                // leds[1].setOff();
                // Utils.sleep(100);
                // leds[1].setOn();
                break;

        case Opcodes.RETURN:

                callStack.pop();
                if (runningProcess.stackEmpty()) {
                        runningProcess = null;
                } else {
                        Object returnValue = runningProcess.pop();
                        if (!callStack.empty()) {
```

```
                                        runningProcess = (RunningProcess) callStack
                                                        .firstElement();
                                        runningProcess.push(returnValue);
                                } else {
                                        runningProcess = null;
                                }
                        }
                        break;

                case Opcodes.HALT:
                        // System.exit(0);
                        runningProcess = null;
                        callStack.pop();
                        break;

                case Opcodes.CALL:
                        Module callBrane = (Module) runningProcess.pop();
                        String callFunctionName = (String) runningProcess.pop();

                        Function callFunction = callBrane
                                        .getFunctionByName(callFunctionName);

                        int callNumberParameters = ((Integer) runningProcess
                                        .pop()).intValue();
                        Vector callParameters = new Vector();
                        for (int callIndex = 0; callIndex < callNumberParameters; callIndex++) {
                                callParameters.addElement(runningProcess.pop());
                        }
                        runningProcess.setProgramCounter(1);
                        RunnableProcess callRunnableProcess = new RunnableProcess(
                                        callFunction);
                        for (int callIndex = 0; callIndex < callNumberParameters; callIndex++) {
                                callRunnableProcess.push(callParameters
                                                .elementAt(callIndex));
                        }
                        runningProcess = new RunningProcess(callRunnableProcess);
                        callStack.push(runningProcess);
                        break;

                default:
                        // should not happen, for compiler debug

                }

        }
    }
});
/**
 * This thread is responsible to receive and deserialize the messages
 * that other sensors in network have sent.
 */
incomingThread = new Thread(new Runnable() {
        public void run() {
                while (true) {
                        try {

                                // create radiogram connection to receive broadcast
                                RadiogramConnection connectionToReceive =
                                     (RadiogramConnection) Connector
                                                .open("radiogram://:100");
                                // create radiogram to receive something

                                Datagram datagramReceive = connectionToReceive
                                                .newDatagram(connectionToReceive
                                                                .getMaximumLength());

                                connectionToReceive.receive(datagramReceive);

                                // read the datagram
                                String first = datagramReceive.readUTF();
                                // get address to create connection
                                String address = datagramReceive.getAddress();
                                // response to broadcast
                                if (first.equals("ping")) { // BCAST case
                                        System.out.println("BROADCAST HANDSHAKE - BEGIN");
                                        System.out.println("PING RECEIVED");
                                        // create's the radiogram
```

```
                                        RadiogramConnection connectionToSend =
                                            (RadiogramConnection) Connector
                                                    .open("radiogram://" + address + ":101");
                                        Datagram datagramSend = connectionToSend
                                                    .newDatagram(connectionToSend
                                                            .getMaximumLength());
                                        datagramSend.writeUTF("pong");
                                        connectionToSend.send(datagramSend);
                                        System.out.println("PONG SENT");
                                        System.out.println("BROADCAST HANDSHAKE - END");
                                        StreamConnection streamConnection = (StreamConnection) Connector
                                                    .open("radiostream://" + address + ":50");
                                        // create stream

                                        DataInputStream dataInputStream = streamConnection
                                                    .openDataInputStream();
                                        Utils.sleep(2000);
                                        byte[] bytesFromStream = new byte[dataInputStream
                                                    .available()];
                                        dataInputStream.readFully(bytesFromStream, 0,
                                                    dataInputStream.available());
                                        // deserialize's the stream into a closure

                                        if (bytesFromStream.length > 0) {
                                                Vector message = (Vector) Serializer
                                                            .deSerialize(bytesFromStream);
                                                incomingQueue.put(message);

                                        }
                                        System.out.println("MESSAGE RECEIVED");
                                }
                        } catch (Exception e) {

                        }
                }
        }
});
/**
 * This thread is responsible to send the messages to the other sensors
 * in network.
 */
outgoingThread = new Thread(new Runnable() {
        public void run() {
                try {
                        while (true) {
                                if (!outgoingQueue.isEmpty()) {
                                        // get closure to be serialized
                                        Vector message = (Vector) outgoingQueue.get();
                                        // serializes the closure
                                        byte[] messageSerialized;
                                        messageSerialized = Serializer.serialize(message);

                                        // SEND
                                        // create broadcast connection
                                        RadiogramConnection connectionToSend =
                                            (RadiogramConnection) Connector
                                                    .open("radiogram://broadcast:100");
                                        Datagram datagramSend = connectionToSend
                                                    .newDatagram(connectionToSend
                                                            .getMaximumLength());
                                        // broadcast to receive response from spots
                                        datagramSend.writeUTF("ping");
                                        connectionToSend.send(datagramSend);

                                        // reset the datagram and close's the broadcast
                                        // connection
                                        datagramSend.reset();
                                        connectionToSend.close();
                                        // create radiogram connection to receive all
                                        // broadcast responses
                                        RadiogramConnection connectionToReceive =
                                            (RadiogramConnection) Connector
                                                    .open("radiogram://:101");

                                        Datagram datagramReceive = connectionToReceive
                                                    .newDatagram(connectionToReceive
                                                            .getMaximumLength());
```

```
                                        // create address list of spots in area that
                                        // responded to broadcast
                                        connectionToReceive.receive(datagramReceive);
                                        Vector addressList = new Vector();
                                        // get all spots in area

                                        if (datagramReceive.readUTF().equals("pong")) {

                                                addressList.addElement(datagramReceive
                                                                .getAddress());
                                        }

                                        // create a stream connection and send's the closure
                                        // to all spots addressList
                                        for (int i = 0; i < addressList.size(); i++) {
                                                String address = (String) addressList
                                                                .elementAt(i);
                                                StreamConnection streamConnection =
                                                    (StreamConnection) Connector
                                                                .open("radiostream://" + address
                                                                                + ":50");
                                                DataOutputStream dataOutputStream = streamConnection
                                                                .openDataOutputStream();
                                                try {
                                                        dataOutputStream.write(messageSerialized);
                                                } catch (IOException ioe) {
                                                        System.out.println("ERROR");
                                                }
                                                dataOutputStream.flush();
                                                System.out.println("MESSAGE SENT");
                                        }
                                }
                        }
                } catch (Exception e) {
                        // e.printStackTrace();
                }
        }
    });
}


/**
 * Search a function in the functions placed at brane.
 *
 * @param brane
 * @param functionName
 * @return function
 */
public Function searchFunctionInModule(Module brane, String functionName) {

        Function function;
        Vector functions = brane.getFunctions();
        Enumeration functionsElements = functions.elements();
        while (functionsElements.hasMoreElements()) {
                function = (Function) functionsElements.nextElement();
                if (function.getName().equals(functionName)) {
                        return function;
                }
        }
        return null;

}

/**
 * Extract all the program elements from the byte-code.
 *
 * @param byteCode
 * @return prog
 */
private Program extractProgramInformation(byte[] byteCode) {

        // points directly to the magic number's offset
        int offset = 1;
        // next 4 bytes correspond to the value of magic number
        int magicNumber = byteArrayToInt(new byte[] { byteCode[offset + 3],
                        byteCode[offset + 2], byteCode[offset + 1], byteCode[offset] },
```

```
                    0);
offset += 5;
// points directly to the version number offset
byte version = byteCode[offset];
offset += 2;
// points directly to the number of modules offset
byte numberOfModules = byteCode[offset];


// create variables to be used during the extraction of the info
Vector modules = new Vector();
byte numberOfFunctions;
byte open;
// points to the MODULE tag of the first or only module of the program
offset += 1;
// extract the modules (numberOfModules)
for (int i = 0; i < numberOfModules; i++) {
        // create the module
        Module module = new Module();

        // jumps from MODULE tag for the number of functions offset
        offset += 1;
        // extract number of functions from the offset
        numberOfFunctions = byteCode[offset];


        if (numberOfFunctions != 0) {
                // jumps from number of functions offset to the FUNCTION tag
                offset += 1;
                // extract functions (numberOfFunctions)
                for (int j = 0; j < numberOfFunctions; j++) {
                        // create the function
                        Function function = new Function();


                        // Function Name
                        byte nameSize;
                        // jumps from FUNCTION tag to the name size offset
                        offset += 2;
                        // extract the name size
                        nameSize = byteCode[offset];
                        // create the buffer for the function name
                        StringBuffer functionNameBuffer = new StringBuffer();
                        // jumps to the initial offset of the function name
                        offset += 1;
                        for (int k = 0; k < nameSize; k++) {
                                functionNameBuffer.append((char) byteCode[offset]);
                                offset += 1;
                        }
                        String name = new String(functionNameBuffer.toString());
                        function.setName(name);

                        // jumps from LOCALS tag to the number of locals value
                        // offset
                        offset += 1;
                        // number of locals
                        byte numberOfLocals = byteCode[offset];
                        function.setNumberOfLocals(numberOfLocals);
                        // jumps from number of locals offset to the initial
                        // byte-code size offset
                        offset += 2;
                        // byte-code itself
                        int sizeOfByteCode = byteArrayToInt(new byte[] {
                                        byteCode[offset + 3], byteCode[offset + 2],
                                        byteCode[offset + 1], byteCode[offset] }, 0);
                        // jumps to the initial offset of the byte-code
                        offset += 4;
                        // buffer for the byte-code
                        StringBuffer byteCodeBuffer = new StringBuffer();



                        // extract the byte-code
                        for (int k = 0; k < sizeOfByteCode; k++) {
                                byteCodeBuffer.append((char) byteCode[offset]);
                                offset += 1;
                        }
```

```
                    byte[] functionByteCode = byteCodeBuffer.toString()
                                    .getBytes();
                    function.setByteCode(functionByteCode);
                    // jump to the number of symbols offset
                    offset += 1;
                    // extract the number of symbols
                    byte numberOfSymbols = byteCode[offset];



                    // jump to the first symbol
                    offset += 1;
                    // create the vector symbols
                    Vector symbols = new Vector(numberOfSymbols);
                    for (int k = 0; k < numberOfSymbols; k++) {
                            switch (byteCode[offset]) {

                            case Tags.BOOL:
                                    symbols.addElement(new Byte(byteCode[offset + 1]));
                                    offset += 2;
                                    break;

                            case Tags.FLOAT:
                                    symbols.addElement(new Float(byteArrayToFloat(
                                                    new byte[] { byteCode[offset + 4],
                                                            byteCode[offset + 3],
                                                            byteCode[offset + 2],
                                                            byteCode[offset + 1] }, 0)));
                                    offset += 5;
                                    break;

                            case Tags.INTEGER:
                                    symbols.addElement(new Integer(byteArrayToInt(
                                                    new byte[] { byteCode[offset + 4],
                                                                    byteCode[offset + 3],
                                                                    byteCode[offset + 2],
                                                                    byteCode[offset + 1] }, 0)));
                                    offset += 5;
                                    break;

                            case Tags.STRING:
                                    offset += 1;
                                    byte stringSize = byteCode[offset];
                                    offset += 1;
                                    StringBuffer stringInBytesBuffer = new StringBuffer();
                                    for (int l = 0; l < stringSize; l++) {
                                            stringInBytesBuffer
                                                            .append((char) byteCode[offset]);
                                            offset += 1;
                                    }
                                    symbols.addElement(stringInBytesBuffer.toString());
                                    break;

                            default:
                                    break;
                            }
                            function.setSymbols(symbols);
                    }
                    // get the functions of module and add the new one
                    Vector functions = module.getFunctions();
                    functions.addElement(function);
                    module.setFunctions(functions);
                    // offset += 1;

            }
    }
    // jump the offset of TAG open
    offset += 1;
    // define the TAG open value
    module.setOpen(byteCode[offset]);
    offset += 1;
    modules.addElement(module);

}
```

```
        offset += 2;
        byte numberOfTypes = byteCode[offset];

        Vector types = new Vector();
        if (numberOfTypes != 0) {
                for (int i = 0; i < numberOfTypes; i++) {
                        int stringSize = byteCode[offset + 1];
                        byte[] stringInBytes = null;
                        offset += 2;
                        StringBuffer stringBufferTypes = new StringBuffer();
                        for (int j = 0; j < stringSize; j++) {
                                stringBufferTypes.append((char) byteCode[offset]);
                                offset += 1;
                        }
                        types.addElement(new String(stringInBytes));
                        offset += 1;

                }
        }

        return new Program(magicNumber, version, types, modules, byteCode);
}

/**
 * Convert the byte array to an int starting from the given offset.
 *
 * @param b
 * @param offset
 * @return value
 */
protected static int byteArrayToInt(byte[] b, int offset) {
        int value = 0;
        for (int i = 0; i < 4; i++) {
                int shift = (4 - 1 - i) * 8;
                value += (b[i + offset] & 0x000000FF) << shift;
        }
        return value;
}

/**
 * Convert the byte array to an float starting from the given offset.
 *
 * @param b
 * @param offset
 * @return float
 */
protected static float byteArrayToFloat(byte[] b, int offset) {

        int i = 0;
        int len = 4;
        int cnt = 0;
        byte[] tmp = new byte[len];
        for (i = offset; i < (offset + len); i++) {
                tmp[cnt] = b[i];
                cnt++;
        }
        int accum = 0;
        i = 0;
        for (int shiftBy = 0; shiftBy < 32; shiftBy += 8) {
                accum |= ((long) (tmp[i] & 0xff)) << shiftBy;
                i++;
        }
        return Float.intBitsToFloat(accum);
}

/**
 * Returns the queue of events to process.
 *
 * @return runQueue
 */
public Queue getRunQueue() {
        return runQueue;
}

/**
 * Returns the program's magic number.
 *
```

```
         * @return magicNumber;
         */
        public int getMagicNumber() {
                return magicNumber;
        }

        /**
         * Returns the program's brane.
         *
         * @return brane
         */

        public Module getBrane() {
                return sensorInterface;
        }



        void printSymbols(Vector symbols) {
                Enumeration symbolsElements = symbols.elements();
                int i = 0;
                while (symbolsElements.hasMoreElements()) {
                        System.out.println("symbols[" + i + "]: "
                                        + symbolsElements.nextElement());
                }

        }

        protected void destroyApp(boolean arg0) throws MIDletStateChangeException {
                // TODO Auto-generated method stub

        }

        protected void pauseApp() {
                // TODO Auto-generated method stub

        }

        protected void startApp() throws MIDletStateChangeException {


                try {

                        byte[] byteCode = Code.byteCode; // the bytecode
                        vm = new VirtualMachine();
                        vm.execute(byteCode);

                } catch (MalformedByteCodeException e) {
                }
        }
}
```

Bibliography

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[2] Jennifer Yick, Biswanath Mukherjee, Dipak Ghosal. Wireless sensor network survey. In *Computer Networks*, volume 52, pages 2292–2330, August 2008.

[3] F. Martins, L. Lopes, and J. Barros. Towards the safe programming of wireless sensor networks. In *Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11. ACM Press, 2003.

[5] The TinyOS Documentation Project. Available at `http://www.tinyos.org`.

[6] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. *Local Computer Networks, Annual IEEE Conference on*, 0:455–462, 2004.

[7] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[8] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85–95. ACM Press, 2002.

[9] P. Levis, D. Gay, and D. Culler. Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines. Technical Report UCB//CSD-04-1343, University of California at Berkeley, August 2004.

[10] R Newton, A. Massachusetts, M Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. *Information Processing in Sensor Networks*, 2005.

[11] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *First International Workshop on Data Management for Sensor Networks (DMSN'04), Toronto, Canada*, 2004.

[12] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine. In *Proceedings of VEE'06*. ACM Press, June 2006.

[13] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a practical Smalltalk Written in Iself. In *Proceeding of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*. ACM Press, October 1997.

[14] Mote Runner White Paper. Available at `www.zurich.ibm.com`.

[15] L. Lopes, F. Martins, J. Barros. Programming Wireless Sensor Networks. In B. Garbinato, H. Miranda, L. Rodrigues, editor, *Middleware for Network Eccentric and Mobile Applications*, pages 25–41. Springer-Verlag, 2009.

[16] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 23th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004.

[17] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services (MobiSys'04)*, June 2004.

[18] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[19] K. Whitehouse, C. Sharp, D. Culler, and E. Brewer. Hood: A Neighborhood Abstraction for Sensor Networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys'04)*, 2004.

[20] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming Wireless Sensor Networks using Kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, June 2005.

[21] B. Greenstein, E. Kohler, and D. Estrin. A Sensor Network Application Construction Kit (SNACK). In *Proceedings of the 2nd International Conference on Embedded Sensor Systems (SENSYS'04)*, pages 69–80, 2004.

[22] P. Stanley-Marbell, C. Borcea, K. Nagaraja, and L. Iftode. Smart Messages: A System Architecture for Large Networks of Embedded Systems. In *Position summary in the Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[23] Borcea, C. and Intanagonwiwat, C. and Kang, P. and Kremer, U. and Iftode, L. Spatial programming using smart messages: Design and implementation. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 690–699, 2004.

[24] L. Lopes, F. Martins, M. S. Silva, and J. Barros. A Process Calculus Approach to Sensor Network Programming. In *Proceedings of the International Conference on Sensor Technologies and Applications (SENSORCOMM'07)*. IEEE Press, 2007.

[25] R. Milner. A Calculus of Communicating Systems. Number 92 in LNCS. Springer-Verlag, 1980.

[26] Tiago Cogumbreiro. A note on the concrete syntax for Callas, 2009. Available at `http://www.dcc.fc.up.pt/callas`.

[27] Eclipse Development Team. Eclipse platform. Available at `www.eclipse.org`.