Rui Miguel Ferreira Mendes

Experiments with the Callas Programming Language and its Virtual Machine

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Setembro de 2012

Rui Miguel Ferreira Mendes

Experiments with the Callas Programming Language and its Virtual Machine

Dissertação submetida à Faculdade de Ciências da Universidade do Porto como parte dos requisitos para a obtenção do grau de Mestre em Ciência de Computadores

Orientador: Prof. Luís Lopes

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Setembro de 2012

Acknowledgements

First of all, I would like to thank my supervisor Prof. Luís Lopes, not only for the support given in the work during this thesis, but also for the support given during the entire research scholarship where I have been since 2010.

A special thanks to my girlfriend for all the support given in every single aspect of my life, and also to my family for the trust and strength I needed to reach this far in my life.

Further, I would like to thank all my colleagues who accompanied me during these six university years for all the help given whenever necessary.

An important acknowledgment to Sr. Melo and D. Estela for all the good times spent in the now-defunct Casa Melo.

Last but not least, I would like thank all my friends who know the true meaning of the string "Tá Quente" for all the fun moments who helped make these six years unforgettable.

Resumo

Redes de sensores sem fios (WSN) são sistemas distribuidos compostos por pequenos dispositivos capazes de fazer medições de determinadas grandezas ambientais, processar dados e comunicar entre si. Estes dispositivos são, tipicamente, muito limitados em termos de CPU, memória e recursos de energia. Juntando o facto de que as WSN são maioritariamente desenhadas para aplicações muito específicas e devem ser capazes de se auto-configurar em aplicações com um grande número de sensores, faz com que a programação deste tipo de redes seja uma tarefa difícil.

Atualmente existe um número considerável de linguagens de programação para WSN que fornecem niveis distintos de percepção do *hardware* e da rede. Algumas delas forçam o utilizador a programar praticamente ao nível do *hardware*, mas outras usam compiladores sofisticados para abstrair dos sensores e da infra-estrutura de rede. Apesar disso, todas elas falham num importante aspecto. Nenhuma é baseada num modelo para descrever computações numa rede de sensores, que permita que propriedades da linguagem, o seu ambiente de execução e as suas aplicações sejam demonstradas.

A linguagem de programação Callas para WSN é baseada num modelo formal que garante propriedades como "type-safety" e "soundness" do sistema de execução. Isto traduz-se numa garantia de segurança para as aplicações no sentido em que erros comuns na execução podem ser evitados por implementação ou estaticamente em tempo de compilação. A linguagem usa um formato de *bytecode* personalizado e corre-o numa máquina virtual criada para esse propósito.

Nesta tese apresentamos uma série de experiências com a linguagem de programação Callas e a sua máquina virtual, com o objectivo de verificar a sua usabilidade e expressividade através da introdução de novos construtores, novas abstracções, modificações no modelo adjacente, e optimizando o consumo de recursos. Relatamos as nossas descobertas fornecendo a base lógica e os detalhes da implementação, e exemplificando as novas características com exemplos escritos em Callas.

Abstract

Wireless sensor networks (WSN) are distributed systems composed of small devices capable of sensing the environment, processing data, and communicating with each other. These devices are typically very limited in terms of CPU, memory, and power resources. Joining the fact that WSN are mostly designed for very specific applications, and must be able to self-configure in large-scale sensor applications, makes the task of programming this type of networks very challenging.

Currently exists a considerable number of programming languages for WSN providing distinct levels of hardware and network awareness. Some of them force the user to program basically at the hardware level, but others use sophisticated compilers to abstract away from the sensor devices and from the network infra-structure. Nevertheless, all of them fail in one important aspect. None is based on a model for describing computations in sensor networks allowing properties of the language, its runtime and applications to be demonstrated.

The Callas programming language for wireless sensor networks is based on a formal model that allows properties like type-safety and soundness of the runtime system to be proved. This translates into a safety assurance for applications in the sense that common runtime errors can be prevented by design or statically at compile time. The language uses a custom bytecode format and runs it on a virtual machine created for this purpose.

In this thesis we present a series of experiments with the Callas programming language and its virtual machine with the goal of verifying its usability and expressiveness through the introduction of new constructs, new abstractions, changes to the underlying model, and optimizing the resource footprint. We report our findings providing the design rationale and implementation details, and exemplifying the new features with examples written in Callas.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Wireless Sensor Networks (WSN) can be viewed as collections of small, low-cost sensor devices communicating over wireless channels [1]. The sensor network is normally divided into two groups: nodes and basestation (also called sink). The nodes are the devices capable of sensing the environment and can be deployed over wide areas. The basestation device is connected to a computer and allows the programmer to collect and manipulate data. Applications in WSN are manifold and embrace different areas [23]. A small overview of these possibilities can be found in Table 1.1.

WSN differ substantially from other mobile ad-hoc networks. The main differences are: (a) they are mostly designed for very specific applications; (b) the sensor nodes have very limited CPU, memory, and power resources; and (c) they must be able to self-configure in large-scale sensor applications without human intervention [14]. All these limitations make the task of programming WSN very challenging.

Currently there are several programming languages for WSN that provide distinct levels of hardware and network abstraction. In PushPin [13], for example, the hardware abstraction is absent and the user is required to program directly at the hardware level. The language uses pieces of native code that are able to interact with each other using a shared memory address space to communicate. Protothreads [7] for the Contiki [6] operating system, is a more elaborated computation model, allowing the creation of multi-threaded applications while maintaining the resources consumption low. Another low-level language is the component-based nesC [11]. This language makes use of the operating system TinyOS [21], calling functions from its modules.

Higher in the level of abstraction we have the macro-programming languages. This type of languages uses sophisticated compilers to abstract away from the hardware

Areas	Applications
Military	Enemy tracking Biological and Chemical Attack Detection
Health	Patient Monitoring Assisted Living
Biology	Animal Tracking Animal and Plant Monitoring
Environment	Fire Detection Flood Detection Pollution Levels
Management	Traffic Control Inventory Control
Industry	Factory Monitoring Machine Monitoring Gas Concentrations
Home	Home Automation (Domotics)

Table 1.1: Overview of typical WSN applications

or from the network infra-structure. One example of a macro-programming language is the TinyDB [16] language, where the programmer sees the sensor network as a database, using SQL-like queries to obtain data from the devices. Similarly, there are other languages like Cougar [9] that use a database model as an abstraction for the sensor network. There are also languages which abstract the sensor network using mobile agents (e.g., Sensorware [4] and Agilla [8]), streams (e.g., Regiment [18]) or regions (e.g., Abstract Regions [22] and Kairos [12]). In [14] we have a comprehensive review of programming languages for wireless sensor networks.

1.1 Problem Statement and Motivation

Current WSN programming languages are designed top-down by mapping high-level constructs from their programming model into low-level runtimes, e.g., implemented in nesC. This approach is prone to errors since: (a) there is often no underlying formal model for the language; and (b) the semantics may not be preserved from the top level constructs to their low-level implementations.

Callas [15] is a programming language based on a formal model that aims to be correct by design. The model allows properties of the language and its runtime to be demonstrated. Furthermore, it is a type-safe language and its runtime is sound. Type-safety means that well-typed applications will not violate the semantics of the programming language. The soundness of the runtime ensures that it will not incorrectly execute applications and produce runtime errors.

The motivation for this thesis emerged from the need of evaluating the usability, the expressiveness and, in some aspects, the efficiency of the language. There were several questions that were raised becoming the basis for this thesis:

- "Can we seamlessly implement common patterns in WSN applications with Callas?";
- "Can we make programming easier by introducing new abstractions or extending existing ones?";
- "Can we make the runtime system more efficient while preserving its soundness?".

With these questions in mind, some changes to the language and to the model were thought and more demanding applications were tested in Callas. Here we present a detailed account of these experiments.

1.2 Contributions

In order for the work in this thesis to be possible it was mandatory the studying of the Callas formal model specification and its compiler and virtual machine implementation. It was also necessary an exploratory interaction with the sunSPOT [20] devices, and the learning of the sunSPOT SDK needed to program those devices, since the language runtime currently only supports this platform. After this learning process, several work on the language was made, and the main contributions are:

- evaluation of the expressiveness of Callas through examples;
- a more friendly syntax;
- changes to the communication model and introduction of channel abstractions;

- changes to the timer model and a more efficient implementation;
- new abstract syntax and virtual machine specification.

1.3 Outline

In Chapter 2, we present the Callas programming model, its syntax and bytecode created to run on a virtual machine. Chapter 3 contains the description of the Callas virtual machine, presenting its data structures and state transitions for every instruction. The contributions for this thesis are described in Chapter 4 and 5. Chapter 4 explains the rationale and implementation of the changes, and Chapter 5 the new syntax and semantics for the Callas programming language. Chapter 6 contains a description of the implementation of a large demo created in Callas using the new syntax, which use a prediction model to minimize the communication in a sensor network application. Finally, Chapter 7 concludes this thesis with some final considerations.

Chapter 2

The Callas Programming Language

Callas [15, 17] is a calculus for programming WSN that aims to establish a formal framework upon which programming languages, runtime systems, and, ultimately, applications may be built safe by design, in the sense that they can be statically guaranteed not to produce runtime errors. Working upwards from this calculus and its static and operation semantics, a programming language and a matching virtual machine were designed and implemented. The language is type-safe and the runtime preserves the semantics of the language, a property also known as soundness. In this chapter we present the language syntax and the compilation process into Callas bytecode.

2.1 Language Syntax

The Callas language syntax is inspired in the Python programming language, using the white space to delimit blocks. We will use a simple application written in Callas to present its syntax. This application reads the temperature from the network every second for one minute. Figure 2.1 shows the file main.calnet which contains information

```
1 interface = iface.caltype
2 externs   = sunspot.caltype
3
4 sensor:  code = sink.callas
5 sensor:  code = sampler.callas
```

Figure 2.1: The top level project file: main.calnet.

about a Callas project. The interface attribute is used to specify the file that provides the type for the application. The externs attribute is used to specify the file that contains the type of the target devices, which in this case are the SunSPOT [20] devices (Figure 2.2).

```
1  defmodule Extern:
2      bool   setLEDColor(long pos, long red, long green, long blue)
3      bool   setLEDOn(long pos, bool isOn)
4      bool   logLong(long val)
5      bool   logDouble(double val)
6      bool   logString(string val)
7      string macAddr()
8      long   getTime()
9      long   battLevel()
10     long   getLuminosity()
11     double getTemperature()
12     double getAccelX()
13     double getAccelY()
14     double getAccelZ()
15     double getAccel()
16     double getInclX()
17     double getInclY()
18     double getInclZ()
```

Figure 2.2: The hardware interface: sunspot.caltype.

The attribute sensor contains references to the files that implement the behavior for each type of node when the application is deployed. In this specific case, we have code for two different types of devices: sinks and samplers. For each instance of sensor, the Callas compiler uses the initial attributes to produce a corresponding byte-code file, e.g., sink.calbc and sampler.calbc.

Figure 2.3 presents the code for the iface.caltype. For this application we defined three main modules where Nil represents the empty module. Each module provides the interfaces for the functions it implements. An instance of module Sensor, in this case, must implement two functions: logData and listen, plus the functions in module Deploy which it extends. All of the defined modules can be used freely as arguments or return values in the interfaces of functions and may also be extended. For example, function deploy has an argument which is a module of type Sampler.

The code for both sink and sampler nodes must implement the application interface.

```
1  defmodule Sampler:
2      Nil sample()
3      Nil run()
4
5  defmodule Deploy:
6      Nil deploy(Sampler sampler)
7
8  defmodule Sensor(Deploy):
9      Nil logData(string mac, double temp)
10     Nil listen()
```

Figure 2.3: The application type: iface.caltype.

In some cases a function may not be required by one of the sides, in which case its body is composed of the process **pass**. Figure 2.4 represents the code for the sink. The first instructions to be executed in a Callas application are the ones outside the modules, i.e, the top-level instructions, and therefore the application starts by storing the module logger in the sensor memory (line 24), and then sends a deploy message with the module sampler as the argument (line 25). The communication is made through a preset channel, that uses the primitive radiogram protocol, since this application was developed for the SunSPOT devices. When sending a message the nodes always send it to the sink, and the sink always sends it to all nodes. The last instruction implements a timer that runs the function listen every second for one minute (line 26).

Analyzing the rest of the code, the first module defined is the sampler that implements two functions: run and sample. Function run simply creates a timer that runs the sample every second for one minute (line 5). This last function queries the hardware for the MAC address and temperature (lines 7-8), and sends a logData message with that information (line 9). Actually, this module contains the code that will run on the samplers, so the destination of the message is the sink.

The other module present in the code is the logger. This module has two main functions defined, since the function deploy is not required in the sink side. The function logData is a simple function that prints the information received as argument (lines 14-20). The samplers will use this function as message to send the MAC address and temperature value. The last function, listen, is used to receive incoming messages (lines 21-22).

In the samplers side the code is quite simple (Figure 2.5). We create a timer using listen to try to receive any message sent by the sink. The deploy is the main function for the sensing nodes, since it receives a code module as argument and runs it, by

```
1  from iface import *
2
3  module sampler of Sampler:
4    def run(self):
5        self.sample() every 1000 expire 60000
6    def sample(self):
7        mac  = extern macAddr()
8        temp = extern getTemperature()
9        send logData(mac,temp)
10
11 module logger of Sensor:
12   def deploy(self, sampler):
13       pass
14   def logData(self, mac, temp):
15       extern logString("\nMAC address: ")
16       extern logString(mac)
17       extern logString("\nTemperature: ")
18       extern logDouble(temp)
19       extern logString("Celsius")
20       pass
21   def listen(self):
22       receive
23
24 store logger
25 send deploy(sampler)
26 listen() every 1000 expire 60000
```

Figure 2.4: The code for the sink side: sink.callas.

calling run function.

After deploying the application to the sunSPOTs, the application works as follow: the sink sends the code to be executed to the sensing nodes (the module sampler). The samplers, that immediately start to listen on the channel, receive the message and run the code carried within. This code starts producing logData messages, which are routed by the underlying radiogram protocol up to the sink where they are handled by the function with the same name, that prints the data items. The samplers continue to produce data, and the sink printing it as it arrives for sixty seconds.

```
1  from iface import *
2
3  module logger of Sensor:
4      def deploy(self, code):
5          code.run()
6      def logData(self, mac, temp):
7          pass
8      def listen(self):
9          receive
10
11 store logger
12 listen() every 1000 expire 60000
```

Figure 2.5: The code for the sensing nodes: sampler.callas.

2.2 The Concrete Syntax

In the previous section we presented the Callas syntax using an example. Now, we will take a look at the concrete syntax of the language to better understand how the language is structured. Figure 2.6 contains the Callas concrete syntax. A program, p, is defined by a vector of type definitions, $\vec{d}$, specifying the application interface, followed by a vector of terms, $\vec{t}$, containing the implementation of the application. A type definition, d, is composed by a keyword, **defmodule**, followed by a type identifier, T, and a set of function signatures, $\vec{s}$. These function signatures contain the information about the methods that compose this type definition, and maintain information about the type of the function, τ, as well as the number and type of the parameters, $\vec{a}$, needed for the function. There are four types in the language: integer, float, boolean, and a type identifier. This last one can be any type defined earlier in the application. A term is also divided in four possibilities. It can be an assignment, $x = e$; it can be a module, M, that uses a name, x, and its type, T, followed by a vector of functions, $\vec{f}$.A function is defined using a label, l, a set of arguments, $\vec{x}$, and a vector of terms, $\vec{t}$. A term might also be a conditional, **if** v : ¶ $\vec{t}$ **else** : ¶ $\vec{t}$, or an expression, e.

There are different type of expressions in Callas: (a) a value, v, that carries the data possible to be exchanged between sensors. It can be a variable, x, an integer, a boolean or a floating point, (b) a unary or binary operation, where it uses one or two values, v, respectively, and performs the desired operation, (c) a **load** or a **store** operation: the first gets the module installed in the sensor, and the second stores

p	$::= \vec{d}\,\vec{t}$	*Programs*
d	$::=$ **defmodule** $T : \P\ \vec{s}$	*Type Defs.*
s	$::= \tau\ l(\vec{a})\P$	*Func. Sigs.*
a	$::= \tau\ x$	*Typed Params.*
τ	$::=$	*Types*
	int	integer
	| **float**	float
	| **bool**	boolean
	| T	type identifier
t	$::=$	*Terms*
	$x = e\ \P$	assign
	| M	module
	| $e\ \P$	expression
	| **if** $v : \P\ \vec{t}$ **else** $: \P\ \vec{t}$	conditional
M	$::=$ **module** x **of** $T : \P\ \vec{f}$	*Modules*
f	$::=$ **def** $l(\vec{x}) : \P\ \vec{t}$	*Functions*

e	$::=$	*Expressions*
	v	value
	| unop v	unary op.
	| v binop v	binary op.
	| **load**	load
	| **store** v	store
	| $v \,\|\|\, v$	merge modules
	| $v.l(\vec{v})$	function call
	| **extern** $l(\vec{v})$	external call
	| $l(\vec{v})$ **every** v **expire** v	timed call
	| **send** $l(\vec{v})$	communication
	| **receive**	communication
v	$::=$	*Values*
	x	variable
	| ... | 0 | ...	integer
	| **True** | **False**	boolean
	| ... | 0.0 | ...	floating point

The symbol ¶ represents the end-of-line character.

Figure 2.6: The syntax of Callas.

value, v, as the main module for the application, (d) a merge of two modules, $v \,||\, v$, (e) a call $v.l(\vec{v})$ to a function, where v represents the module of the function l with $\vec{v}$ as the arguments to the function, (f) an external call, **extern** $l(\vec{v})$, to functions that are hardware dependent. Whenever, we need to measure some physical property (e.g, luminosity) that interacts directly with the hardware, this type of calls is used, (g) a timed call, $l(\vec{v})$ **every** v **expire** v, used to call an installed function $l(\vec{v})$ periodically, (h) a remote call, **send** $l(\vec{v})$, used to place a message $\langle l(\vec{v})\rangle$ into the output queue, and (i) a receiving process, **receive** , used to get a message from the input queue and add the packaged function call, $l(\vec{v})$, to the run queue, where awaits for execution.

From the concrete syntax, it is possible to create an abstract syntax specifically to sensors. The abstract syntax allows us to define the operational semantics of the language resorting in some reduction rules [5]. In parallel, there is a type system

which restricts the valid programs of the language. This, added to the operational semantics, proves the type-safety of the language. Also, the type-safety property along with the specification of the virtual machine, that will be presented later in this thesis, guarantees the soundness of the virtual machine.

2.3 Compilation and Deployment

The Callas programming language uses a virtual machine to run its applications. But, in order to run it, each of the files that compose a Callas project are first compiled to a custom bytecode format. Figure 2.7 shows the runtime representation of the bytecode. The top level of the bytecode is the program, $\mathcal{P}$, and contains an array of module definitions. A module definition, $\mathcal{D}$, is a map of strings, containing the function names, onto tuples representing the functions bodies. A function body, $\mathcal{F}$, is a tuple containing three integers that hold the number of parameters, of free variables and of local variables for the function. It also contains a bytecode array, $\mathcal{B}$, holding the code for the function, and an array, $\mathcal{U}$, that carries the constants that occur in the source code for the function. The virtual machine works with basic data types (boolean, integer, float) and with modules, $\mathcal{M}$.

program	$\mathcal{P} \in \textsc{ArrayOf}(\mathcal{D})$
module bytecode	$\mathcal{D} \in \textsc{MapOf}(\textsf{String} \mapsto \mathcal{F})$
function bytecode	$\mathcal{F} \in \textsf{Int} \times \textsf{Int} \times \textsf{Int} \times \mathcal{B} \times \mathcal{U}$
values	$\mathcal{U} \in \textsc{ArrayOf}(v)$
function code	$\mathcal{B} \in \textsc{ArrayOf}(c)$
value	$v \in \textsf{Bool} \cup \textsf{Int} \cup \textsf{Float} \cup \mathcal{M}$
module	$\mathcal{M} \in \textsc{MapOf}(\textsf{String} \mapsto \mathcal{F} \times \textsc{ArrayOf}(v))$
instruction	$c \in \{\textsf{loadm}\ i, \textsf{loadm2}\ i, \textsf{call}, \textsf{call2}, \textsf{merge}, \textsf{send}, \textsf{receive}, \textsf{timer}, \textsf{return},$ $\textsf{jmp}\ i, \textsf{ift}\ i, \textsf{loadb}, \textsf{storeb}, \textsf{loadc}\ i, \textsf{load}\ i, \textsf{store}\ i, \textsf{dup}, \textsf{swap}$ $\mathit{binop}, \mathit{unop}\}$

Figure 2.7: The byte-code format.

Modules are dynamic instances of module definitions since they store, for each of the function in the module definition, the free variables together with the instructions,

c, for the function body. The virtual machine is stack based allowing a compact instruction set with few addressing modes. Thus, operands must be on top of the operand stack in order to be used, and for that the virtual machine has load and store instructions to move values between the function environment and the operand stack. The instruction-set also contains instructions for handling modules, making calls, receiving or sending values through the network, control-flow and a set of arithmetic and logic operations.

The compilation process (Figure 2.8) is mainly composed by two steps. In the first step, the source code for each file is transformed into a corresponding Callas bytecode file. Then, the second step embeds the byte-code files in a .jar or .suite file, depending on the target platform, at a specific point within the package hierarchy. This packet is then deployed in the target platform and executed using sunSPOT specific commands. At the time the applications starts to execute, e.g. startApp() for SunSPOT MIDlets, the virtual machine loads the embedded byte-code into run-time data structures which are used thereafter.

Figure 2.9 shows a textual representation of the file node.callas from the application previously presented in order to better understand the translation from a Callas file to the respective bytecode. The figure highlights the different methods implemented in Callas. Each function contains the information about the parameters, free variables and local variables. Every symbol needed for the function is placed after its respective bytecode, e.g., the definition of a module Nil and a string run after the code for the deploy function. Previous to the description of the methods in the logger module, we have the anonymous top-level module containing the bytecode for the instructions outside the modules followed by a list of the symbols used in the code.

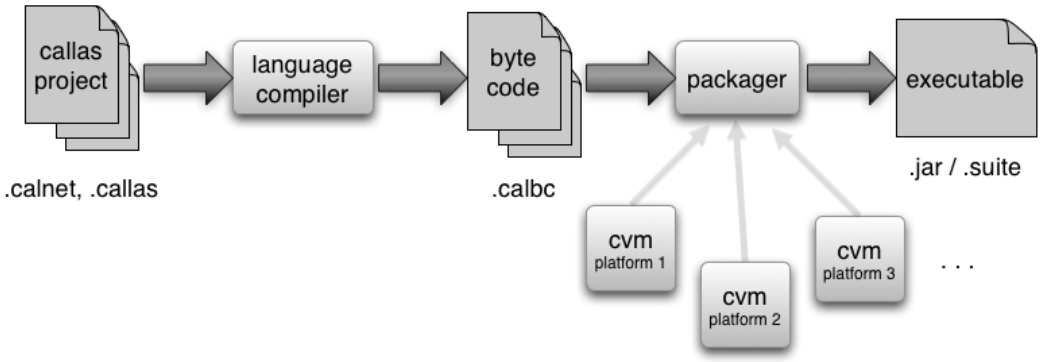

Figure 2.8: The compilation process.

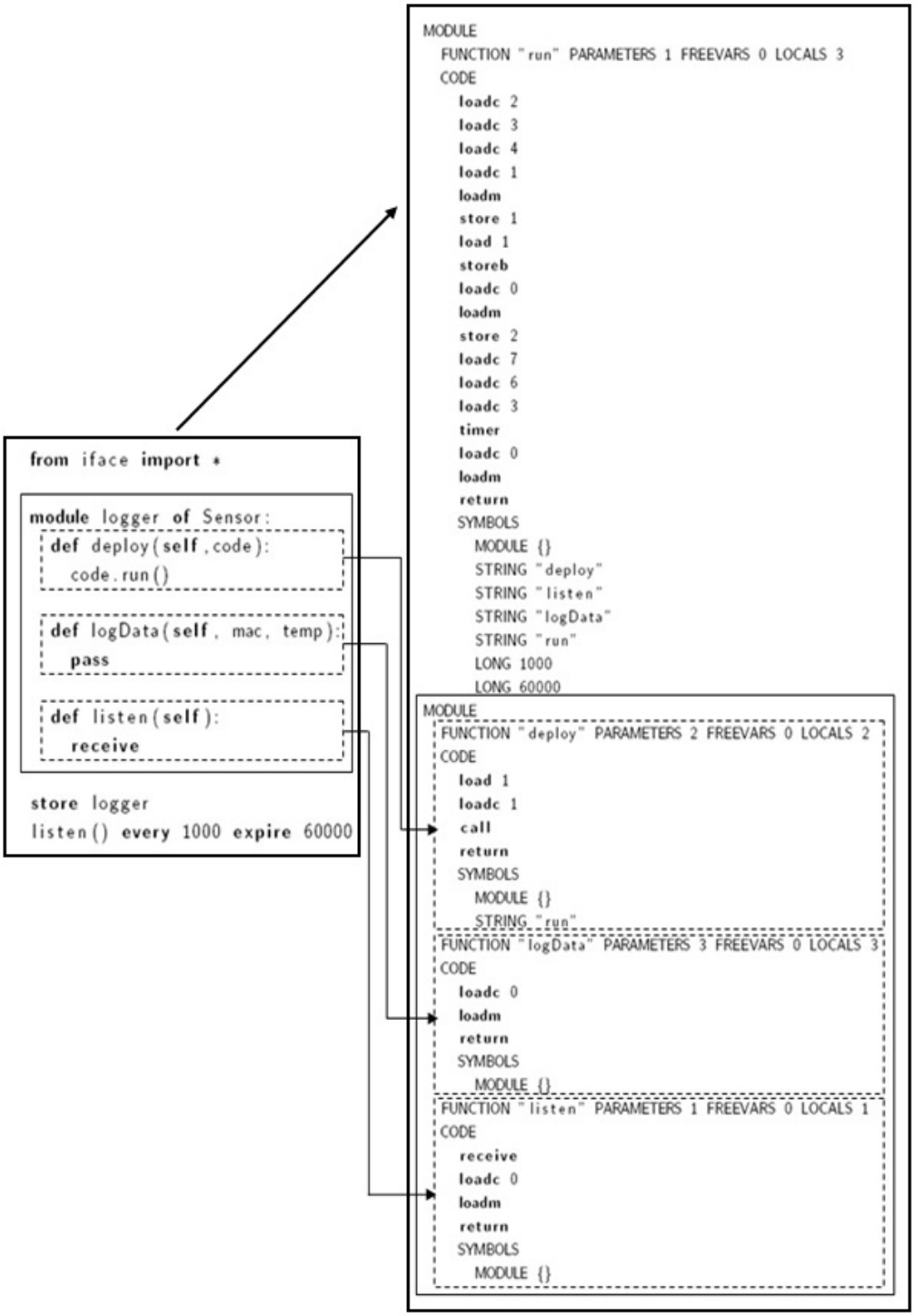

Figure 2.9: Bytecode translation of file node.callas

Completed the description of the Callas programming language syntax and bytecode, the next chapter overviews the Callas virtual machine, its associated data structures, and explains the bytecode execution.

Chapter 3

The Callas Virtual Machine

The Callas programming language uses a custom bytecode format, and therefore in order to interpret it, a matching virtual machine is required. In this chapter we will describe in detail the Callas virtual machine that provides an abstraction from the hardware and allows the execution of Callas applications.

3.1 Data Structures

The virtual machine contains a set of data structures (Figure 3.1) that are used to represent its state, $\mathcal{G}$. The state is given by a pair $\mathcal{P}, \langle \mathsf{Int}\,, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R} \rangle_{\mathcal{O}}^{\mathcal{I}}$ that represents the bytecode for the program, as described on the previous chapter. The **Int** represents

machine state	$\mathcal{G} \in$	$\mathcal{P} \times \mathsf{Int} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{I} \times \mathcal{O}$
timers	$\mathcal{T} \in$	$\textsc{SetOf}(l(\vec{v}) \times \mathsf{Int} \times \mathsf{Int} \times \mathsf{Int}\,)$
call-stack	$\mathcal{C} \in$	$\textsc{StackOf}(\mathsf{Int} \times \mathcal{E} \times \mathcal{S} \times \mathcal{B} \times \mathcal{U})$
waiting calls	$\mathcal{R} \in$	$\textsc{QueueOf}(l(\vec{v}))$
messages	$\mathcal{I}, \mathcal{O} \in$	$\textsc{QueueOf}(\langle l, \vec{v} \rangle)$
environment	$\mathcal{E} \in$	$\textsc{ArrayOf}(v)$
operand stack	$\mathcal{S} \in$	$\textsc{StackOf}(v)$

Figure 3.1: The syntactic categories of the virtual machine.

an integer value used as an internal clock for the machine. Next, $\mathcal{M}$ is a module that contains all the functions installed in a sensor. This module can be updated during execution, but it is not possible to add or remove functions, just replace already existing ones. The $\mathcal{T}$ represents a set of timed calls. This type of calls are used to make periodic calls to functions. Each of the timed calls is composed of a pending call, $l(\vec{v})$, and three integers: the first represents the periodicity of the call, the second specifies the time when the timed call expires, and the last one contains the next invocation of the timer. Using an interrupt-like mechanism, whenever it is time for the next invocation, the execution is paused and the respective call is added to the run queue. Continuing with the machine state description, the $\mathcal{C}$ is a call-stack that stores call-frames which are composed of a program counter, an environment frame $\mathcal{E}$, an operand stack $\mathcal{S}$, a bytecode array $\mathcal{B}$, and a constant array $\mathcal{U}$. The environment frame is an array that stores the values for the parameters, the free variables and the local variables of a function. The $\mathcal{R}$ represents the run queue used to store pending calls. They wait the return of the current call, before being loaded onto the call-stack. The last data structures of the virtual machine state are the input and output queue, $\mathcal{I}$ and $\mathcal{O}$ respectively, which are used to interact with lower layers of the sensor network protocol stack in order to receive and send, respectively, messages through the network.

3.2 Initial State

The initial state of the virtual machine is obtained by loading the bytecode on the program, $\mathcal{P}$. Every program $\mathcal{P}$ contains an anonymous module with a function run, at offset 0, containing top-level instructions in the original program. This function has no parameters or free variables. Thus, loading a program $\mathcal{P}$ is done by a function boot() obtaining the following result:

$$\mathcal{P}, \langle 0, \mathcal{M}_0, \{\}, (0, \epsilon, \epsilon, \langle \mathsf{loadc}\ 0, \mathsf{call2}, \mathsf{return} \rangle, \langle \text{"run"} \rangle), \epsilon \rangle_{\epsilon}^{\epsilon} \leftarrow \mathsf{boot}(\mathcal{P})$$

The function loads a representation of the top level module, $\mathcal{M}_0$, into the virtual machine and installs a piece of bytecode that starts the program by calling run. In order to load the identifier for the function from the constant array, the bytecode starts by using the instruction loadc 0 and then uses the instruction call2 to call the respective function. The return instruction simply ends the bytecode sequence and

allows calls from the run queue to be loaded to the call-stack. In the initial state of the virtual machine, the input and output queues, run queue, and set of timed calls are empty.

3.3 Transition Rules

In this section we will analyze the state transitions for the main instructions, which can be called by the virtual machine. In what follows, we will use a set of tables containing three fields: $\mathcal{B}[i]$, assumptions, and transitions. The $\mathcal{B}[i]$ represents the instruction inspected by the virtual machine. The transitions will contain a description of the new state provided that the assumptions are satisfied. Table 3.1 contains the state transitions for the instructions that allow the loading and merging of modules, and the calling of functions. In terms of loading modules, the virtual machine has two possible instructions: loadm and loadm2. In both cases, they receive an index j to identify the module in the bytecode that is required to load. This operation needs some preparatory work. The map containing the bytecode for the module is collected in $\mathcal{P}[j] = \{l_k \mapsto \mathcal{F}_k\}_{k \in I}$, where $I = \{0 \dots n\}$ is a set of consecutive integer indexes, l_k is the function name, and $\mathcal{F}_k = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$ is a tuple containing information about function l_k, that includes the number of parameters j_1, free variables j_2, local variable j_3, and its bytecode and array of constants. If none of the functions in the module has free variables, the instruction used is the loadm and it results in a simple closure for the module. Otherwise, it is used the loadm2 instruction and the resulting module includes arrays of values, acquired from the current environment, which represent the free variables for each of the module's functions. These values are provided on top of the operand stack, listed by function.

Calling a function also uses two different instructions: call and call2. While call handles the functions that interact with the hardware (device sensors and actuators), the call2 handles the calls to functions implemented by the application programmer. In the first case, the function name l, and the arguments to the call $\vec{v}$ are placed at the top of the operand stack and consumed. Then, a built-in function of the virtual machine that acts as interface with the underlying operating system or with a library, handles the call. The virtual machine maintains internal information about the possible system calls and therefore uses the information about the arity of the function l to prepare the call to $\mathsf{sysCall}(l, \vec{v})$. The value returned by the system call is placed at top of the operand stack. The call2 instruction needs the name of the function l, the respective

Table 3.1: Transition rules for function calls and handling modules instructions

$\mathcal{B}[i]$	Assumptions	Transitions
loadm j	$\mathcal{P}[j] = \{l_k \mapsto \mathcal{F}_k\}_{k \in I}$	$t \to t'$ (time)
	$\forall_k, \mathcal{F}_k = (\_, 0, \_, \mathcal{B}, \mathcal{U})$	$i \to i + 2$ (instruction pointer)
		$\mathcal{S} \to \mathcal{S} : \{l_k \mapsto (\mathcal{F}_k, \epsilon)\}_{k \in I}$ (operand stack)
loadm2 j	$\mathcal{P}[j] = \{l_k \mapsto \mathcal{F}_k\}_{k \in I}$	$t \to t'$
	$\mathcal{F}_k = (\_, j_k, \_, \mathcal{B}, \mathcal{U})$	$i \to i + 2$
	$j_k = \lvert\vec{v}_k\rvert$	$\mathcal{S} : \vec{v}_n : l_n : \cdots : \vec{v}_0 : l_0 \to \mathcal{S} : \{l_k \mapsto (\mathcal{F}_k, \vec{v}_k)\}_{k \in I}$
call	$\lvert\vec{v}\rvert = \mathsf{arity}(l)$	$t \to t'$
	$v = \mathsf{sysCall}(l, \vec{v})$	$i \to i + 1$
		$\mathcal{S} : \vec{v} : l \to \mathcal{S} : v$
call2	$\mathcal{M}(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}', \mathcal{U}')$	$i \to i + 1$
	$\mathcal{E}' = \langle \mathcal{M} \vec{v}_1 \vec{v}_2 \vec{0} \rangle$	$\mathcal{C} : (i, \mathcal{E}, \mathcal{S} : \vec{v}_1 : l : \mathcal{M}, \mathcal{B}, \mathcal{U})$ (call stack)
	$j_1 = \lvert\vec{v}_1\rvert$, $j_3 = \lvert\vec{0}\rvert$	$\to \mathcal{C} : (i + 1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : (0, \mathcal{E}', \epsilon, \mathcal{B}', \mathcal{U}')$
return		$t \to t'$
		$i \to i + 1$
		$\mathcal{C} : (i', \mathcal{E}', \mathcal{S}', \mathcal{B}', \mathcal{U}') : (i, \mathcal{E}, \mathcal{S} : v, \mathcal{B}, \mathcal{U})$
		$\to \mathcal{C} : (i', \mathcal{E}', \mathcal{S}', \mathcal{B}', \mathcal{U}')$
merge	$\mathcal{M}_3 = \mathsf{merge}(\mathcal{M}_1, \mathcal{M}_2)$	$t \to t'$
		$i \to i + 1$
		$\mathcal{S} : \mathcal{M}_2 : \mathcal{M}_1 \to \mathcal{S} : \mathcal{M}_3$

module $\mathcal{M}$, and the arguments to the call $\vec{v}_1$, placed at top of the operand stack. While consuming these values, the instruction collects runtime information about $\mathcal{M}$ and l that includes the bytecode for the function $\mathcal{B}'$, the function's constants $\mathcal{U}'$, the values for the free variables $\vec{v}_2$, and the size of the environment frame $j_1 + j_2 + j_3$. With the arguments to the call, $\mathcal{M}\vec{v}_1$, along with the values of the free variables $\vec{v}_2$ and extra space for the local variables, $\vec{0}$, with size j_3, a new environment frame $\mathcal{E}$ is built. Finally, the instruction uses that information to create a new call-frame, with program counter at 0 and an empty operand stack, and pushes it on top of the call stack $\mathcal{C}$. The return instruction is used to transfer values from one call-frame to another. The virtual machine expects a value, v, on top of the operand stack of the call-frame which is executing. Then, the current call-frame is removed from the call

Table 3.2: Transition rules for communication and timed calls instructions

$\mathcal{B}[i]$	Assumptions	Transitions
send	$\mathcal{M}_0(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$	$i \to i + 1$
	$j_1 = \lvert\vec{v}_1\rvert$	$\mathcal{S} : \vec{v}_1 : l \to \mathcal{S}$
		$\mathcal{O} \to \langle l, \vec{v}_1 \rangle :: \mathcal{O}$ (outgoing queue)
receive		$t \to t'$
		$i \to i + 1$
		$\mathcal{I} :: \langle l, \vec{v} \rangle \to \mathcal{I}$ (incoming queue)
		$\mathcal{R} \to l(\vec{v}) :: \mathcal{R}$ (run queue)
timer	$\mathcal{M}_0(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$	$i \to i + 1$
	$j_1 = \lvert\vec{v}_1\rvert$	$\mathcal{S} : j : \vec{v}_1 : l \to \mathcal{S}$
		$\mathcal{T} \to \mathcal{T} \cup \{(l(\vec{v}_1), j, k, t + j)\}$ (timers)
(interrupt)	$t = t'$	$\mathcal{T} \uplus \{(l(\vec{v}), j, k, t')\} \to \mathcal{T} \cup \{(l(\vec{v}), j, k, t' + j)\}$
		$\mathcal{R} \to l(\vec{v}) :: \mathcal{R}$

stack and the value v is placed on top of the operand stack associated to the next call-frame on the call stack.

The instruction merge is important, since it allows the dynamic update of code modules. This instruction expects two modules, $\mathcal{M}_1$ and $\mathcal{M}_2$, of the same type at the top of the operand stack and as a result puts a new module, $\mathcal{M}_3$, that merges both modules, on top of the operand stack.

Table 3.2 has a second set of instructions for the virtual machine. This set contains the communication instructions send and receive and the instruction timer, used to deal with periodic tasks. The send starts by inspecting the definition of function l in the module $\mathcal{M}_0$, in order to obtain its arity. With this information the virtual machine knows how many values are needed to fetch from the operand stack, and using the function name and the call arguments, a message, $\langle l, \vec{v} \rangle$, is built and added to the end of the output queue, for further processing. Another instruction used for communication is the receive. In this case, the virtual machine checks the input queue for messages. If a message is found, it takes the message on top of the queue and puts it on the run queue, where it waits for execution as a pending call. The message contains all the information needed to build the pending call. If no message is found

Table 3.3: Transition rules for arithmetic and control flow instructions

$\mathcal{B}[i]$	Assumptions	Transitions
binop	$v_3 = binaryOperation(v_1, v_2)$	$t \to t'$ $i \to i+1$ $\mathcal{S} : v_2 : v_1 \to \mathcal{S} : v_3$
unop	$v_2 = unaryOperation(v_1)$	$t \to t'$ $i \to i+1$ $\mathcal{S} : v_1 \to \mathcal{S} : v_2$
dup		$t \to t'$ $i \to i+1$ $\mathcal{S} : v \to \mathcal{S} : v : v$
swap		$t \to t'$ $i \to i+1$ $\mathcal{S} : v_2 : v_1 \to \mathcal{S} : v_1 : v_2$
jmp j		$t \to t'$ $i \to i+j$
ift j	$k = \begin{cases} i+j, & v = true \\ i+1, & v = false \end{cases}$	$t \to t'$ $i \to k$ $\mathcal{S} : v \to \mathcal{S}$

in the input queue, the instruction returns and the program continues unaffected by the receive.

The timer instruction is used to program periodic tasks. This instruction involves storing the call, $l(\vec{v}_1)$, the period, j, the expire value, k, and the next invocation, $t+j$, in a tuple in the array of timers, $\mathcal{T}$. In order to extract the number of values necessary from the stack, the instruction inspects in $\mathcal{M}_0$, the definition of the function l. The tasks are triggered using an interrupt-like mechanism at a given instant t'. Using this interrupt the execution pauses, the respective pending call $l(\vec{v})$ is added to the bottom of the run queue, and the execution resumes. The next invocation of the timer, $t'+j$, is also updated.

Table 3.3 shows another set of instructions of the virtual machine. Starting by the binaryOperation, where it pops from the operand stack the two values needed

Table 3.4: Transition rules for data management instructions

$\mathcal{B}[i]$	Assumptions	Transitions
loadc j	$v = \mathcal{U}[j]$	$t \to t'$ $i \to i+1$ $\mathcal{S} \to \mathcal{S} : v$
loadb	$\mathcal{G} = (\mathcal{P}, j, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R}, \mathcal{I}, \mathcal{O})$	$t \to t'$ $i \to i+1$ $\mathcal{S} \to \mathcal{S} : \mathcal{M}$
storeb	$\mathcal{G}' = (\mathcal{P}, j, \mathcal{M}', \mathcal{T}, \mathcal{C}, \mathcal{R}, \mathcal{I}, \mathcal{O})$	$t \to t'$ $i \to i+1$ $\mathcal{S} : \mathcal{M}' \to \mathcal{S}$ $\mathcal{G} \to \mathcal{G}'$
load j	$v = \mathcal{E}[j]$	$t \to t'$ $i \to i+1$ $\mathcal{S} \to \mathcal{S} : v$
store j	$\mathcal{E}'[j] = \begin{cases} \mathcal{E}[k], & k \neq j \\ v, & k = j \end{cases}$	$t \to t'$ $i \to k$ $\mathcal{S} : v \to \mathcal{S}$

for the operation (v_1 and v_2), executes the respective binary operation and then puts the result, v_3, on top of the operand stack for further use. Similar to the binaryOperation, we have the unaryOperation, except in the number of arguments needed for the operation. In this case, only one value is expected on top of operand stack. The instruction dup consists in duplicate the value on top of the operand stack. Thus, it is only needed to pop the value, v, from the stack and then push the same value twice. A swap is viewed in the virtual machine as the swap between the two values on top of the operand stack. The implementation is quite simple, it pops the two values from the stack and pushes them in the reverse order. The last two instructions make jumps in the program bytecode. The jmp has an integer as argument, j, and simply adds that value to the program counter. The ift instruction expects a boolean value on top of the operand stack and if that value is true the j is added to the program counter; if not, the program counter continues to the next instruction.

Finally, Table 3.4 contains the last set of instructions for the virtual machine. Instruc-

tion loadc is used to collect a value from the constant array, $\mathcal{U}$. It has an integer, j, as argument, containing the index of the value on the constant array. Thus, value v is fetched and placed on top of the operand stack. Next in the table, we have the instructions to load and store code in the sensor. The loadb simply places module $\mathcal{M}$ from the machine state on top of the operand stack. The storeb executes the opposite, expecting a module, $\mathcal{M}'$, on top of the operand stack and storing it as the main module on the machine state. The final two instructions are the load and store, and are used to load or store values in a given index of the environment frame, $\mathcal{E}$. Both instructions have an integer, j, as argument, that represents the related environment frame index. The load fetches the value v from the environment frame and places it on top of the operand stack. The store executes the opposite, expecting a value v on top of the operand stack and storing it on the position j of the environment frame.

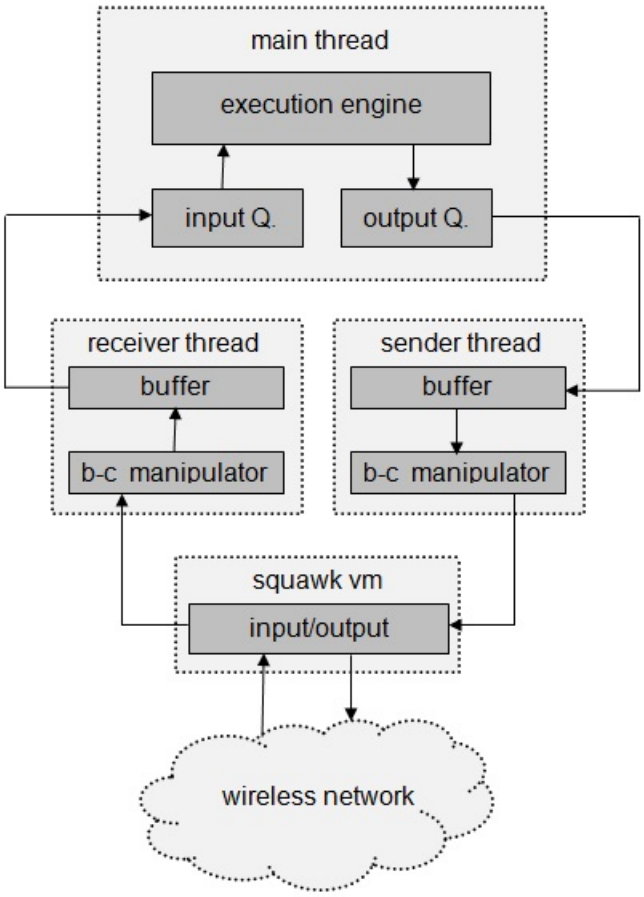

Figure 3.2: Architecture of the Callas virtual machine

The Callas virtual machine is implemented in Java and uses the sunSPOT SDK to

interact with the sunSPOT devices. The bytecode interpreter simply has a main loop, which runs a switch statement with end case an instruction operand and its implementation. Figure 3.2 shows the virtual machine architecture. It is divided in three threads: (a) the main thread, who is responsible for the interpretation of the program bytecode, trigger timed calls and the interaction with the communication threads; (b) the receiver thread, used to receive messages from the network, unpack them and create the function calls which are consumed by the main thread; (c) and the sender thread, that consumes function calls, packs them into bytecode messages and broadcast them. All the low-level network communication is handled by the Squawk virtual machine [19], on top of which runs the Callas virtual machine when using the sunSPOT devices.

With all the virtual machine instructions described in this section we now understand how the Callas programming language works and the steps needed from the programming of an application to its execution in the virtual machine. The next chapter explains in detail my contributions for this project.

Chapter 4

Building on Callas

In this chapter we describe the problems we detected when working with Callas as a programming tool. We will present each problem and the solution we have implemented in three steps: (a) the problems, (b) changes to the compiler, and (c) changes to the virtual machine.

4.1 A Program is a Module

Problem: In order to simplify the language we modified it in a way that programs in Callas now have a single top level module with the method init implemented. Before this modification we could have a number of modules defined independently, and top-level instructions outside those modules that were compiled into a special internal module. When translating the instructions to the bytecode format, it was needed to add a new anonymous module that contained all the instructions outside the modules. These were the first instructions to be executed in the virtual machine. To improve this process, the programmer now is forced to create a single top level module that has all the initial instructions within a method named init. With this change we simplify the translating process and the organization of the code in Callas. We also address a problem in the semantics of the virtual machine which would make its initial state not typable. For example, the program in Figure 4.1 would crash, because it was necessary for the programmer to store the main module. If an instruction store was missing in the code for an application, the virtual machine would not have any module defined in its state, and therefore would not recognize any function call.

```
module app of Sensor:
  def print(self):
    extern logString(``This text will never be printed'')

print() every 1000 expire 5000
```

Figure 4.1: Example of a demo that would crash in the previous version of Callas.

Compiler: The main changes for this modification were made in the compiler. The first one in the parser. Previously, the language accepted more than one module and instructions outside those modules. As shown in Figure 4.2, a program in Callas could

```
program ::=
      stmts:s
        {: RESULT = parser.composeBody(s); :}
    | typeregs
        {: RESULT = new Code(parser.getLocation(0, 0)); :}
    | /* empty */
        {: RESULT = new Code(parser.getLocation(0, 0)); :}
    ;
```

Figure 4.2: Old version of the Callas parser top level condition.

be a stmts that represents any possible set of instructions made in Callas, or simply a typeregs that refers to the definition of types and imports. The program could also be an empty program. In order to force the existence of a single top level module this production was changed (Figure 4.3). Now, a Callas program cannot be seen as

```
program ::=
      typeregs moduledef:m
        {:
        LinkedList ll = new LinkedList();
        ll.add(m);
        RESULT = parser.composeBody(ll); :}
    | typeregs
        {: RESULT = new Code(parser.getLocation(0, 0)); :}
    | /* empty */
        {: RESULT = new Code(parser.getLocation(0, 0)); :}
```

Figure 4.3: Current version of the Callas parser top level condition.

a set of instruction of any kind. The stmts were replaced by a typeregs, that can be

empty, followed by a definition of a module, moduledef. Thus, a program is limited to only one top level module. After updating the syntax, the compiler checks that an

```
1  public CodeType adapt(String filename, Map<Symbol, RecursiveType> types)
2   throws ErrorMessagesException {
3    RecursiveType type = types.get(Symbol.symbol("Sensor"));
4    if (type == null) {
5      throw new ErrorMessagesException(new SyntacticError(
6       new SourceLocation(filename), "Type 'Sensor' that declares the
7       sensor type is not declared."));
8    }
9    CodeType sensorType = (CodeType) type.type;
10   TypeEquality typeEq = new TypeEquality();
11   Map<Symbol, FunctionType> funcs = new TreeMap<Symbol, FunctionType>
12    (sensorType.functions);
13   if (!funcs.containsKey(Symbol.symbol("init")))
14     throw new ErrorMessagesException(new SyntacticError(new
15      SourceLocation(filename), "Function 'init' in module Sensor
16       is missing."));
17   if (!funcs.get(Symbol.symbol("init")).isNilType())
18     throw new ErrorMessagesException(new SyntacticError(new
19      SourceLocation(filename),"Function of signature 'Nil init' in
20       module Sensor is missing."));
21   return (CodeType) typeEq.unfold(new RecursiveType(type.variable,
22    new CodeType(funcs)));
23  }
```

Figure 4.4: Parser function responsible for the sensor type parse.

init method of signature Nil init() is present in the top-level module of type Sensor. Figure 4.4 shows the code for function adapt of the type checker, responsible for this verification. In lines 3-8 it searches for a module of type Sensor, which needs to be declared as the top-level module. If it does not exist an error message is returned. Then, in lines 9-12 the functions present in the module are fetched in order to be able on line 13 to verify if function init exists. Once again, if the function does not exist, an error message is sent (lines 14-15). Also, we need to verify if the function init is of type Nil (line 16), otherwise an error is thrown (lines 17-19). The final modification in the compiler was in the translate package, where previously we needed to create a new module containing all the instructions outside the modules. Now the translation of a program to the Callas bytecode, aside of the definition of types, it is equivalent to translate just one module. Figure 4.5 starts by creating a list of modules present in the application (line 2), and then we simply need to translate the top-level module,

with index 0, since every other modules are implemented inside this one and will be recursively translated.

```
1 public CVMModule compile(CallasProcess from) {
2   List<Code> modules = Functions.modulesProc(from);
3   return translate(modules.get(0));
4 }
```

Figure 4.5: Function in the translate package used to translate a program

Virtual Machine: With the aforementioned modifications in the compiler, the virtual machine only needs, when initiated, to load the module at offset 0, that represents the top level module, and to place a call to the function init on top of the call stack. This way, the virtual machine starts by running the instructions defined in the init function. The modifications also allowed the initial state to be typable, since now it is guaranteed that it will exist a main module installed in the sensor that implements the type of the application. Before this change, the programmer were required to store the main module, because it was not defined by default, and therefore the initial state of the virtual machine would not be typable.

```
1 //(...)
2 Module firstModule = program.create();
3 Function firstFunction = firstModule.lookup("init");
4 setExecuting(new Running(firstFunction, new Object[] { firstModule }));
5 //(...)
```

Figure 4.6: Code for the initialization of the bytecode interpreter

Figure 4.6 shows the code used to initialize the bytecode interpreter. The instruction in line 2 gets the top level module in the Callas application and stores the information about its functions in a runtime representation of a Callas module. Then, using this module, a lookup is made for function init (line 3) and the method setExecuting places a new call-frame, created using the function init and the top-level module, on top of the call stack (line 4).

4.2 Derived Constructs

There were a few programming patterns being used so often that derived constructs really made it easier on the programming and made the concrete syntax cleaner. In this section we present a description of the changes related to the implementation of those derived constructs.

4.2.1 The Operator ||=

Problem: Merging two modules is a common operation in Callas applications. In order to facilitate the programming in Callas we implemented the operator "||=". This operator is used in the form "x ||= y" which is equivalent to "x = x || y".

Compiler: Only small changes are required in the compiler with the creation of the new construct that runs an adaptation of the already existing merge. Figure 4.7 shows the code for both constructs. The existing merge (lines 1-3) simply uses both modules, left and right, as arguments for the process type associated to the merge operation, Update. In the case of the new construct (lines 4-10), we need to use the identifier for the first module, n, to create a new variable that will be used as argument along with the second module for the Update. Finally, the result will be the assignment of the resulting value from the merge operation to the variable, var, that represents the first module.

```
1  | value:left MERGE:i value:right
2      {: RESULT = new Update(parser.getLocation(ileft, iright), left,
3        right); :}
4  | ID:n MERGE_ASSIGN:i value:v
5      {:
6        SourceLocation loc = parser.getLocation(nleft, nright);
7        Variable var = new Variable(loc, n);
8        Update up = new Update(parser.getLocation(ileft,iright),var,v);
9        RESULT = new Assignment(var, up);
10     :}
```

Figure 4.7: Implementation of the "||" and "||=" operators

4.2.2 Install

Problem: A wide range of Callas applications need to install new code in the sensors. That process includes loading the sensor code to a variable, updating it using the merge operation, and storing the new code in the sensor (Figure 4.8). Therefore, it makes sense to implement a single instruction, which can automatically do those three instructions. Thus, we implemented a constructor that takes as argument the code we want to install and encapsulates all three instructions (Figure 4.9).

```
1 def deploy(self,newCode):
2   code = load
3   code ||= newCode
4   store code
```

Figure 4.8: Installing a new code without the install command

```
1 def deploy(self,newCode):
2   install newCode
```

Figure 4.9: Installing a new code with the install command

Compiler: Once again this was not a major change in terms of implementation. The already existing processes in the language were used to create the new constructor install. The processes needed for this were LoadSensorCode, Update, StoreSensorCode and also Let to represent the chain of instructions. In Figure 4.10 we have the parser code for the instruction install. The parser expects a value, v, that represents the module we want to install. In lines 4-5 we create a new variable, var, with an arbitrary name. This variable, ultimately, will represent the resulting module of the entire operation. Then, using the process types needed for this instruction, new processes are created in lines 6-8. At last, in line 9 we create the first chain of instructions, let1, using p2 and p3 that represent, respectively, an Update process and a StoreSensorCode process. Then, the result will be the second chain of instructions, let2, chaining p1, which represents the LoadSensorCode process, with let1 (lines 10-11).

```
1  | INSTALL:x value:v
2    {:
3      SourceLocation loc = parser.getLocation(xleft, xright);
4      String varName = parser.newVariableName();
5      Variable var = new Variable(loc, varName);
6      LoadSensorCode p1 = new LoadSensorCode(loc);
7      Update p2 = new Update(loc,var,v);
8      StoreSensorCode p3 = new StoreSensorCode(loc,var);
9      Let let1 = new Let(var,p2,p3);
10     Let let2 = new Let(var,p1,let1);
11     RESULT = let2;
12   :}
```

Figure 4.10: Implementation of the install construct

4.3 Programmer Defined Channels

Problem: One of the main limitations of the language was its communication model. The virtual machine started with two predefined channels, one to send messages and another to receive. These channels could not be changed by the programmer and it was also not possible to add a different channel, the programmer could only use the predefined channels, where the nodes would always send the messages to the basestation, and the basestation would always send the messages to all nodes. Therefore, it was needed to change this situation, but that required major changes in the language syntax. We needed to create instructions to open and close communication channels and also an interface to manage the different channels opened by the programmer.

```
1  | OPEN:x value:v
2          {: RESULT = new Open(parser.getLocation(xleft, xright), v);  :}
3  | CLOSE:x value:v
4          {: RESULT = new Close(parser.getLocation(xleft, xright), v); :}
```

Figure 4.11: Clauses for instructions open and close in the parser

Compiler: In the compiler there was the need to create two new keywords, open and close. Then, creating a clause in the parser for both instructions (Figure 4.11) and implementing two new process types to represent both instructions in the compiler. These new process types simply store the information about the location of the instruction in the source code and the string representing the channel. We use a string to represent the channel so it could be possible to accept different kind of

```
1  public List<CVMStmt> caseOpen(Open open) {
2          List<CVMStmt> stmts = new LinkedList<CVMStmt>();
3          stmts.addAll(compile(open.channel));
4          stmts.add(CVMOpen.OPEN);
5          stmts.addAll(compile(Code.NIL));
6          return stmts;
7          }
8
9  public List<CVMStmt> caseClose(Close close) {
10         List<CVMStmt> stmts = new LinkedList<CVMStmt>();
11         stmts.addAll(compile(close.channel));
12         stmts.add(CVMClose.CLOSE);
13         stmts.addAll(compile(Code.NIL));
14         return stmts;
15         }
```

Figure 4.12: Implementation of instructions open and close in the Process Translator

communication protocols by a simple analysis to the string content. Different protocols have diverse options, like the definition of the number of hops used in the channel, and in this way it is possible to encapsulate these options in the string itself. The virtual machine must be able to parse this string and create an appropriate channel. The next step in the implementation of open and close is the verification of the type of the argument received by the parser, since in both instructions the argument needs to be of type string. Finally, the last change in the compiler was the creation of two new methods to translate these new instructions to the bytecode. Figure 4.12 contains the code used for the translation. Both instructions are similar, their argument is compiled and placed in a linked list, stmts, then the respective virtual machine instruction is also added, and finally, a Nil is added representing the return value of this instruction.

Virtual Machine: In this case there were also major changes in the virtual machine. The obvious ones are: the creation of two new types in the virtual machine (CVMOpen and CVMClose); and linking two new opcodes with these instructions. In order to manage the channels the program wants to open or close, a new interface named ConnectionManager was created. This interface contains two hashtables, inputConnectionMap and outputConnectionMap, to store the channels, which are divided into input or output channels. The string that represents the channel is the key to the hashtable and the value is a NetworkInputInterface in a case of an input channel, or a NetworkOutputInterface in the case of an output channel. These interfaces allow us to manipulate the different types of channels. Figure 4.13 shows the implementation of

```
1  public class NetworkOutputInterface implements INetworkOutputInterface {
2    RadiogramConnection connectionToSend;
3
4    public NetworkOutputInterface(String str){
5      try{
6        connectionToSend = (RadiogramConnection) Connector.open(str);
7      }catch(Exception e){
8        e.printStackTrace();
9      }
10   }
11   public void send(Call msg) throws IOException{
12     try {
13       Datagram datagramSend = connectionToSend
14       .newDatagram(connectionToSend.getMaximumLength());
15       Serializer ser = new Serializer(datagramSend);
16       ser.writeCall(msg);
17       connectionToSend.send(datagramSend);
18       datagramSend.reset();
19     } finally {}
20   }
21   public void close() throws IOException{
22     connectionToSend.close();
23   }
24 }
```

Figure 4.13: Implementation of the Network Output Interface

the NetworkOutputInterface for the radiogram protocol used by the sunSPOT devices. The class constructor (lines 4-10) expects the string with the channel as argument and simply tries to open it. Then, two additional methods are implemented. The send is used to send messages through the channel and expects a call-frame, msg, as argument. It creates a new datagram (lines 13-14) in order to send the message, serializes the msg (lines 15-16), and finally sends the message through the channel (line 17). The other method, close, simply closes the connection (line 22).

In Figure 4.14 we have the implementation of the interface used to manipulate the input channels (NetworkInputChannel). This interface is run as a separate thread and therefore is slightly different from the previous one. Previously in the language, we only had one input channel and therefore it was only needed an unique queue to store the incoming messages. Now, since we want the programmer to be able to choose from which channel he wants to receive, every NetworkInputInterface will have its own

```
1  public class NetworkInputInterface implements NetworkInterface{
2    private Queue inQueue;
3    private RadiogramConnection connectionToReceive;
4    private boolean isRunning = true;
5    public void run() {
6      inQueue = new Queue();
7      transmit();
8    }
9    public void transmit(){
10     try {
11       Datagram datagramReceive = connectionToReceive
12       .newDatagram(connectionToReceive.getMaximumLength());
13       while (isRunning){
14         connectionToReceive.receive(datagramReceive);
15         if (datagramReceive.getLength() > 0) {
16           Deserializer deser = new Deserializer(datagramReceive);
17           inQueue.put(deser.readCall());
18         }
19         datagramReceive.reset();
20       }
21     }finally{
22       connectionToReceive.close();
23     }
24   }
25   public void setConnection(String str){
26     try{
27       this.connectionToReceive = (RadiogramConnection)Connector.open(str);
28       this.connectionToReceive.setTimeout(10000);
29     }catch(Exception e){
30       e.printStackTrace();
31     }
32   }
33   public Call popCall(){
34     if (inQueue.size() == 0)
35       return null;
36     return (Call) inQueue.get();
37   }
38   public void close(){
39     this.isRunning = false;
40   }
41 }
```

Figure 4.14: Implementation of the Network Input Interface

queue where it stores the received messages, meaning that for every input channel we will have a different queue. When the interface is started, it creates the queue (line 6) and executes the method transmit (line 7). This method is a loop in which we are constantly trying to receive a message and placing it in the respective queue (lines 13-20). The setConnection method defines the channel used by the interface and expects a string with that information as argument. Thus, it opens the channel (line 27) and associates a timeout of ten seconds to it (line 28). The receive method present in the RadiogramConnection is a blocking operation, and therefore this timeout is needed to prevent a deadlock and to be able to terminate the thread. The function popCall is the one used to pop a call from this interface queue. If there are no calls in the queue the method returns the null value. Finally, the close terminates the thread by changing the isRunning variable to false (line 39). With this operation the loop condition (line 13) is no longer true, the channel is closed (line 22) and the thread terminates. Whenever

```
1  public boolean open(String channel){
2    boolean output = parseURL(channel);
3    if (output){
4      INetworkOutputInterface iface = new NetworkOutputInterface(channel);
5      outputConnectionMap.put(channel,iface);
6      return true;
7    } else {
8      NetworkInterface iface = new NetworkInputInterface();
9      iface.setConnection(channel);
10     new Thread(iface).start();
11     inputConnectionMap.put(channel,iface);
12     return true;
13   }
14   return false;
15 }
```

Figure 4.15: Function open from the ConnectionManager class

it is needed to open a channel, the virtual machine expects a string on top of the operand stack (representing the channel) and then uses the ConnectionManager to open it. Figure 4.15 shows the code used in the ConnectionManager to open a channel. It starts by analyzing the string received as argument (line 2). This process is made by the method parseURL that, since we are working with radiogram protocols, simply inspects the string to understand if it is an output or input channel. Then, if it is an output channel, we just create an NetworkOutputInterface and store it in the output related hashtable (lines 4-5).

```
1  public void close(String str) throws IOException{
2    boolean output = parseURL(str);
3    if(output){
4      INetworkOutputInterface nInterface = (INetworkOutputInterface)
5          outputConnectionMap.get(str);
6      nInterface.close();
7      outputConnectionMap.remove(str);
8    }else{
9      NetworkInterface nInterface  = (NetworkInterface)
10         inputConnectionMap.get(str);
11     nInterface.close();
12     inputConnectionMap.remove(str);
13   }
14 }
```

Figure 4.16: Function close from the ConnectionManager class

In the case of an input channel, the code is slightly different since we need to start a new thread to receive messages and store them in a queue for further use by the virtual machine. We create a new NetworkInputInterface (line 8), then define the channel for the connection (line 9), start a new thread with the NetworkInputInterface (line 10), and store the interface in the input related hashtable (line 11). To access the channels afterwards, the ConnectionManager has the methods lookupInput and lookupOutput, that return the respective channel if it exists in the hashtables.

Closing a channel is a simpler process. Likewise the open, the close function expects a string on top of the operand stack. The ConnectionManager simply needs to check if it is an output or input channel, and then get the interface from the respective hashtable, to finally use the method close from the respective interface, to close the channel, and remove it from the memory (Figure 4.16). In the case of an output channel it only closes the connection, but when it is an input channel it also kills the associated thread.

4.3.1 Selecting Channels

Problem: With open and close implemented we required a way to select the channels when sending or receiving messages. So, we introduced a new keyword select, followed by a value identifying the channel that complements the send and receive constructs. With this new syntax the programmer can select on which channel he wants the

program to communicate. In Figure 4.17 we have a small example of a Callas application using this new way of communicate. In this application two channels are opened, for input (line 3) and output (line 4), and then using the new semantics for instructions send and receive, we select the appropriate channel for the communication (lines 6 and 9).

```
1 module sensor of Sensor:
2   def init(self):
3     inputChannel = open "radiogram://:90"
4     outputChannel = open "radiogram://broadcast:90"
5     listen(inputChannel) every 1000
6     select outputChannel send action()
7
8   def listen(self, inputChannel):
9     select inputChannel receive
```

Figure 4.17: Application in Callas using the new communication semantics

Compiler: Everything in the compiler that included the instructions send and receive was affected by this change. We added a new variable to both representations to store the string containing the channel. Figure 4.18 shows the updated production for the instructions send and receive in the parser. As it is possible to see, it now expects the select keyword prior to both instructions and also a value, v, used to represent the channel through which the message will be sent or received. Then, it is necessary to verify if that variable is of the correct type, and adding it in the translation for the bytecode of both send and receive. Figure 4.19 shows the code from the compiler translate package. Both instructions compile the string containing the channel, then add the respective virtual machine instruction, and place the value Nil as the return value (lines 3-6 for send and 11-13 for receive). The only difference is that prior to the compilation of the channel the instruction send also compiles both function name and arguments related to the message that will be sent through the network (line 2).

Virtual Machine: To have the ability to select the channel, some changes were required in the virtual machine. Figure 4.20 shows the updated implementation in the bytecode interpreter for the instructions send and receive. Starting with instruction send, it now pops a string from the operand stack that contains the channel for the communication (line 2) and uses it to lookup for the channel interface in the connection manager, connManager (line 3-4). Next, the name and the arguments of the function we want to send are popped from the operand stack (line 5-6) and then

```
1  | SELECT:t value:v SEND:s ID:function arguments:a
2      {: RESULT = new Send(parser.getLocation(tleft,tright), v, function,
3        a); :}
4
5  | SELECT:t value:v RECEIVE:r
6      {: RESULT = new Receive(parser.getLocation(tleft,tright),v); :}
```

Figure 4.18: Production for instructions send and receive in the parser

```
1  public List<CVMStmt> caseSend(Send send) {
2    List<CVMStmt> stmts = new LinkedList<CVMStmt>();
3    stmts.addAll(compileArgs(send.functionName, send.arguments));
4    stmts.addAll(compile(send.channel));
5    stmts.add(CVMSend.SEND);
6    stmts.addAll(compile(Code.NIL));
7    return stmts;
8  }
9  public List<CVMStmt> caseReceive(Receive recv) {
10   List<CVMStmt> stmts = new LinkedList<CVMStmt>();
11   stmts.addAll(compile(recv.channel));
12   stmts.add(CVMReceive.RECEIVE);
13   stmts.addAll(compile(Code.NIL));
14   return stmts;
15 }
```

Figure 4.19: Functions for instructions send and receive in the process translator

using the channel interface we send the message (line 8). If the channel is not open the instruction will throw an error. The instruction receive also starts by popping the string representing the channel from the operand stack (line 14). Then, using the method popCall from the connManager, the virtual machine tries to get a call from the queue associated to the channel (line 15). If the input queue is empty this instruction does nothing. If a call is retrieved from the queue, then we add the top-level module, installed, as the first arguments of the call and put it on the run queue where it waits for execution (lines 17-18).

4.4 Cleaning Up Timed Calls

Problem: Timed calls are very commonly used in sensor network applications. It is important to optimize and guarantee a good performance with low cost and this was

```
1  public void caseSend(){
2    String channel = popString();
3    INetworkOutputInterface outFace =(INetworkOutputInterface)
4     connManager.lookupOutput(channel);
5    String functionName = popString();
6    Object[] args = popInstalledFunctionParameters(functionName);
7    try {
8      outFace.send(new Call(functionName, args));
9    } catch (IOException e) {
10     e.printStackTrace();
11   }
12 }
13
14 public void caseReceive() {
15   String channel = popString();
16   Call call = connManager.popCall(channel);
17   if (call != null){
18     call.addModule(installed);
19     addRunQueue(call);
20   }
21 }
```

Figure 4.20: Implementation of instructions send and receive in the virtual machine

the main reason to re-define the support for timed calls. Prior to this work, timed calls were supported in the main thread of the virtual machine, by computing the time for the next call. A new approach was required to process timed calls, and since every timed call is independent of each other, the idea was the creation of threads to process them. While threads may be considered expensive for sensor nodes, many operating systems like TinyOS [21] and Contiki [6] support them. Therefore this approach to the timed calls can be adapted for different sensors with distinct architectures and operating systems. Moreover, they allow the clean programming of timers and CPU time to be effectively managed, e.g., putting the main thread to sleep when the run queue is empty, until a timer produces another process. With this approach each thread is responsible for adding the respective call to the run queue and sleeps in between such events.

Virtual Machine: For this change, we needed to change the way the virtual machine handled this type of processes. Thus, we created a new class BoundedTimedTask to represent and handle the timed calls in the virtual machine. This class takes advantage of the package java.util.TimerTask to implement periodic calls (Figure 4.21). Now,

```
1  public class BoundedTimedTask{
2      private Call call;
3      private long period;
4      private Interpreter emu;
5      private Timer timer;
6
7      public BoundedTimedTask(final Call call, long period,
8       final Interpreter emu)
9      {
10       this.call = call;
11       this.period = period;
12       this.emu = emu;
13       this.timer = new Timer();
14     }
15     public void schedule(){
16       timer.scheduleAtFixedRate(new TimerTask() {
17         public void run() {
18                 synchronized(emu){
19                   emu.addRunQueue(call);
20                   emu.notifyAll();
21                 }
22         }
23       }, period, period);
24     }
25
26     public void cancel(){
27       timer.cancel();
28     }
29 }
```

Figure 4.21: Class that implements the timers: BoundedTimedTask.java

when we create a BoundedTimedTask in the virtual machine a new thread is launched, and it automatically puts a new call in the run queue whenever it is time. Note that the run queue of the virtual machine is a thread-safe data structure and these threads are sleeping and only awake to put the calls on the run queue. Also, if the main thread does not have any call in the run queue, but has active timers, it sleeps and only awakes when a timed call occurs. Furthermore, it was necessary to remove the code that was previously responsible for verifying the time and adding the calls to the run queue.

Figure 4.22 shows the code in the virtual machine interpreter to handle the timed calls.

It pops the string containing the function name from the operand stack (line 2), looks up for the function in the top-level module (lines 3-4), fetches the arguments needed for the function and the periodicity value of the timer from the operand stack (lines 5-6) and creates a new BoundedTimedTask, btt (line 7-8). For the creation of this task we need also to create a new call-frame using the function name and its arguments. The other parameters needed are the periodicity of the call and the interpreter itself, to add calls to the run queue. The timed calls are now stored in an appropriate hashtable (timers) and to avoid conflicts a unique name is created and returned by the timer instruction. This unique name (timerName) is created attaching a timetag with the time of its creation to the function name (line 9). The unique name is used as the key in the hashtable timers for the respective task (line 10). Before the instruction is completed we need also to start the thread for the timed call (line 11) and place the timerName on the operand stack in order to be returned (line 12).

```
1  public void caseTimedCall() {
2    String functionName = popString();
3    Module module = peekModule();
4    Function function = module.lookup(functionName);
5    Object[] args = popArray(function.declaration.parametersCount);
6    long period = popLong();
7    BoundedTimedTask btt = new BoundedTimedTask(new Call(functionName,args),
8     period, this);
9    String timerName = functionName + clock.currentTimeMillis();
10   timers.put(timerName, btt);
11   btt.schedule();
12   getExecuting().pushOperand(timerName);
13 }
```

Figure 4.22: Timed calls implementation in the virtual machine interpreter

4.4.1 Adding the Instruction Kill to the Language

Problem: Formerly in the Callas language, whenever a timed call was created it was impossible to access it later either to modify its parameters or to terminate it. The timed call used a parameter expire to determine when to terminate. While re-designing the support for timed calls we decided to implement an useful new construct for the language: kill. This command allows the user to terminate an active timer. Figure 4.23 shows a simple example of the use of the instruction kill, where a timer

is created and terminated after just one iteration. The program starts by creating a timer that runs the method action every second, storing its unique name on the variable timerID (line 3). Then, it stores the timerID in the sensor memory (lines 4-6). When the method action is called for the first time, it prints a message (line 10), loads the timerID stored in the memory (lines 11-12), and terminates the timer (line 13) and consequently the entire program.

```
1  module sensor of Sensor:
2    def init(self):
3      timerID = action() every 1000
4      module temp of timers:
5        def getTimerID(self): timerID
6      install temp
7      pass
8
9    def action(self):
10     extern logString(''Just one iteration'')
11     mem = load
12     timerID = mem.getTimerID()
13     kill timerID
14
15   def timerID(self):
16     ""
```

Figure 4.23: Callas example using the instruction kill

Compiler: The creation of a new construct obviously implies some changes in the compiler. These changes are very similar to the ones related to the instructions open and close, since in all three cases there is a keyword followed by a string. Figure 4.24 contains the implementation of the kill construct in the parser. The command has a single argument (a label that represents the timer) and it is necessary to verify if the type of the argument is a string. Then, it was necessary to create a new type of process in the compiler to represent and store the information about the kill. In order to translate it to the byte-code it was created a new function that translate this new type of process (Figure 4.25). It is created a linked list with the string containing the timer identifier compiled, the instruction kill from the virtual machine and at last the Nil as the return value for the operation.

Virtual Machine: In terms of the virtual machine, it was also necessary to create a new type CVMKill to represent this new command and then associating it to a new opcode. Then, in the bytecode interpreter a new condition was added to handle the

```
1 | KILL:x value:v
2         {: RESULT = new Kill(parser.getLocation(xleft, xright), v); :}
```

Figure 4.24: Implementation of instruction kill in the parser

```
1 public List<CVMStmt> caseKill(Kill kill) {
2   List<CVMStmt> stmts = new LinkedList<CVMStmt>();
3   stmts.addAll(compile(kill.timerID));
4   stmts.add(CVMKill.KILL);
5   stmts.addAll(compile(Code.NIL));
6   return stmts;
7 }
```

Figure 4.25: Implementation of instruction kill in the process translator

kill constructs. Figure 4.26 shows the code used whenever the virtual machine finds a kill command. It pops the string containing the timer identifier from the operand stack (line 2), searches in the hashtable for the respective BoundedTimedTask (line 3), terminates it (line 4), and then removes it from the hashtable (line 5).

```
1 public void caseKill() {
2    String timerID = popString();
3    BoundedTimedTask btt = (BoundedTimedTask) timers.get(timerID);
4    btt.cancel();
5    timers.remove(timerID);
6 }
```

Figure 4.26: Implementation of instruction Kill in the virtual machine

4.4.2 Removing Expire from Timed Calls

Problem: The expire argument from a timer is used to terminate a timer after a period of time. Despite being an useful tool, with the restructuring of the timed calls and the addition of the new constructor kill, expire became an unnecessary argument and it was removed from the definition of timers.

Compiler: The obvious start for this task is the removal of the argument expire from both parser and lexical analyzer. Then, it was also necessary to make some adjustments to: (a) the process used to represent a timed call on the compiler, by removing the variable used to store the expire value; (b) the type verification, where

previously it was checked the expire value, and now is no longer necessary; and (c) the translate part of the compiler, by simply removing the translation of the expire value, only processing the name of the call and period for each timed call.

Virtual Machine: This modification involved the removal of the argument expire in the virtual machine representation of a timed call, and also the removal of the code used to terminate the timer when the expire value was reached. When a timer instruction is encountered in the virtual machine, now it just need to get two arguments from the operand stack instead of three: the name of the function and the value representing the period.

This last change concludes the set of modifications made to the language in the context of this thesis. Obviously, these changes have consequences in the syntax and bytecode presented on Chapter 2 and also in the data structures and state transitions of the virtual machine presented on Chapter 3. Therefore, in the next chapter we will overview the impact of these changes on the Callas programming language and its formal specification.

Chapter 5

The New Callas

The changes mentioned in the prvious chapter had an expected impact in the language. In this chapter we will describe the present state of the language Callas using both Chapter 2 and Chapter 3 of this thesis as a guide to identify the differences. Thus, this chapter will be divided into three sections: language syntax by example; concrete syntax; and virtual machine. The first one will identify the changes in the application example from Chapter 2. The second will present the new concrete syntax. The last one will analyze the differences in the state and behavior of the virtual machine.

5.1 Language Syntax

```
1  defmodule Sampler:
2      Nil sample(string channel)
3      Nil run()
4
5  defmodule Deploy:
6      Nil deploy(Sampler sampler)
7
8  defmodule Sensor(Deploy):
9      Nil init()
10     Nil logData(string mac, double temp)
11     Nil listen(string channel)
```

Figure 5.1: The application type updated: iface.caltype.

In order to perceive the real changes in the syntax of the language we update the

application from Chapter 2 to the new syntax. The top level project file, main.calnet, along with the file containing the hardware interface (sunspot.caltype) remain the same, without any modification. Figure 5.1 shows the first changes in the implementation of the application. Due to the insertion of a top-level module, a function init is added to the Sensor module that will contain the first instructions to be executed (line 9). Also, in the methods listen from the Sensor module and sample from the Sampler module, we add a new argument, that will be used to pass a channel identifier.

```
1  from iface import *
2
3  module logger of Sensor:
4    def init(self):
5      module sampler of Sampler:
6        def run(self):
7          toSink = "radiogram://broadcast:91"
8          open toSink
9          self.sample(toSink) every 1000
10       def sample(self, toSink):
11         mac  = extern macAddr()
12         temp = extern getTemperature()
13         select toSink send logData(mac,temp)
14     toNodes = "radiogram://broadcast:90"
15     fromNodes = "radiogram://:91"
16     open toNodes
17     open fromNodes
18     select toNodes send deploy(sampler)
19     listen(fromNodes) every 1000
20     pass
21
22   def deploy(self, sampler):
23     pass
24   def logData(self, mac, temp):
25     extern logString("\nMAC address: ")
26     extern logString(mac)
27     extern logString("\nTemperature: ")
28     extern logDouble(temp)
29     extern logString("Celsius")
30     pass
31   def listen(self, fromNodes):
32     select fromNodes receive
```

Figure 5.2: The code for the sink updated: sink.callas.

Figure 5.2 contains the updated code for the sink. There were major changes comparing to the code previously presented. The first instructions to be executed now are the ones inside the init function. Thus, the application starts by creating a module sampler (the code to be sent and executed to the nodes) with methods run and sample (lines 5-13). The only difference in the code of sampler is due to the existence of channels in the language. The run method starts by opening a channel to communicate with the sink and passes the channel identifier to the sample function as an argument. In the sample function the only change is the selection of the channel received as argument, when sending the data.

After the creation of the module, the application opens two different channels for sending/receiving messages to/from the nodes (lines 14-17). Next, it uses the output channel (toNodes) to send the module to the nodes (line 18), and creates a timer with the function listen using the input channel (fromNodes) as argument (line 19). The function listen now receives a string as argument that represents the channel identifier and is used to select the channel from where the data is received (lines 31-32). Finally, the logData function, used to print the data, remains unchanged (lines 24-30).

```
1  from iface import *
2
3  module logger of Sensor:
4      def init(self):
5          fromSink = "radiogram://:90"
6          open fromSink
7          listen(fromSink) every 1000
8      def deploy(self, code):
9          code.run()
10     def logData(self, mac, temp):
11         pass
12     def listen(self, fromSink):
13         select fromSink receive
```

Figure 5.3: The code for the sensing nodes updated: sampler.callas.

The code in the sensing nodes required only small changes (Figure 5.3). In the old version of this application, there was only the implementation of a timer using the function listen, outside the modules. Thus, this instruction moves to the function init, with a small change that also occurred in the code for the sink (line 7). Now the function listen needs a string with a channel identifier as argument, and because of this we need to open a channel, fromSink, to receive messages from the sink (lines 5-6).

The rest of the code is similar, since in the deploy method it is only needed to receive a module and call the function run within it.

In short, the application is now more flexible, since the programmer has the possibility to choose through which channel the communication is made. In this application we separate the nodes from the sink using port 90 for communication sent by the sink to the nodes and port 91 for communication in the opposite direction. This type of programming allows, for example, the creation of groups of nodes using different channels to communicate, opening possibilities to a wider number of applications, increasing the expressiveness of Callas.

5.2 Concrete Syntax

Figure 5.4 contains the updated Callas syntax. The notion of a program is now changed. A program is a vector of type definitions followed by a single top-level module. This module is stored in the sensors and must have a function of signature Nil init, which contains the first instructions to be executed.

In terms of expressions, e, three new types were added: (a) the kill process, **kill** v, used to terminate the timer with identifier v; (b) the opening of a communication channel, **open** v, where v represents the link to the channel; and (c) the closing of a communication channel, **close** v, where v, is the link to the channel. These three expressions use a string as argument, so a new type of value, v, was added: strings. Due to the change to the communication semantics, the remote calls are now represented by **select** v **send** $l(\vec{v})$, where channel, v, is selected and message, $\langle l(\vec{v})\rangle$, is sent through that specific channel. The receiving process is now represented by **select** v **receive** , and also uses the channel identifier, v, to fetch a message from the input queue associated to that channel in order to add the packaged function, $l(\vec{v})$, to the run queue, R. The last modification was in the timed call process, where the expire value was removed, being represented only by $l(\vec{v})$ **every** v.

5.3 Virtual Machine

The changes mentioned above had an expected impact on the data structures and transition rules presented in Chapter 3. In addition, the bytecode also suffered a small change associated to the addition of three new constructs to the language, affecting

p	$::= \vec{d}\, M$	*Programs*
d	$::=$ **defmodule** $T : \P\ \vec{s}$	*Type Defs.*
s	$::= \tau\ l(\vec{a})\P$	*Func. Sigs.*
a	$::= \tau\ x$	*Typed Params.*
τ	$::=$	*Types*
	int	integer
	$\mid$ **float**	float
	$\mid$ **bool**	boolean
	$\mid\ T$	type identifier
t	$::=$	*Terms*
	$x = e\ \P$	assign
	$\mid\ M$	module
	$\mid\ e\ \P$	expression
	$\mid$ **if** $v : \P\ \vec{t}$ **else** $: \P\ \vec{t}$	conditional
M	$::=$ **module** x **of** $T : \P\ \vec{f}$	*Modules*
f	$::=$ **def** $l(\vec{x}) : \P\ \vec{t}$	*Functions*

e	$::=$	*Expressions*
	v	value
	$\mid$ unop v	unary op.
	$\mid\ v$ binop v	binary op.
	$\mid$ **load**	load
	$\mid$ **store** v	store
	$\mid\ v \mid\mid v$	merge modules
	$\mid\ v.l(\vec{v})$	function call
	$\mid$ **extern** $l(\vec{v})$	external call
	$\mid\ l(\vec{v})$ **every** v	timed call
	$\mid$ **kill** v	kill timed call
	$\mid$ **open** v	open channel
	$\mid$ **close** v	close channel
	$\mid$ **select** v **send** $l(\vec{v})$	communication
	$\mid$ **select** v **receive**	communication
v	$::=$	*Values*
	x	variable
	$\mid \ldots \mid 0 \mid \ldots$	integer
	$\mid$ **True** $\mid$ **False**	boolean
	$\mid \ldots \mid 0.0 \mid \ldots$	floating point
	$\mid$ "" $\mid \ldots$	string

The symbol ¶ represents the end-of-line character.

Figure 5.4: The syntax of Callas.

the instruction-set of the virtual machine with the inclusion of three new instructions: open, close, and kill.

Figure 5.5 represents the updated data structures of the virtual machine. Starting with the machine state, $\mathcal{G}$, both input and output queues were removed from the state, and it was added instead the connection manager, $\mathcal{N}$, that is now the data structure handling the communication. This connection manager is a map of strings representing the channel identifier into an input queue, $\mathcal{Q}$, associated to the channel. Obviously,

machine state	$\mathcal{G} \in$	$\mathcal{P} \times \textsf{Int} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{N}$
timers	$\mathcal{T} \in$	$\textsc{MapOf}(\textsf{String} \mapsto (l(\vec{v}) \times \textsf{Int} \times \textsf{Int}))$
call-stack	$\mathcal{C} \in$	$\textsc{StackOf}(\textsf{Int} \times \mathcal{E} \times \mathcal{S} \times \mathcal{B} \times \mathcal{U})$
waiting calls	$\mathcal{R} \in$	$\textsc{QueueOf}(l(\vec{v}))$
messages	$\mathcal{Q} \in$	$\textsc{QueueOf}(\langle l, \vec{v} \rangle)$
environment	$\mathcal{E} \in$	$\textsc{ArrayOf}(v)$
operand stack	$\mathcal{S} \in$	$\textsc{StackOf}(v)$
connection manager	$\mathcal{N} \in$	$\textsc{MapOf}(\textsf{String} \mapsto \mathcal{Q})$

Figure 5.5: The syntactic categories of the virtual machine.

in the case of an output channel, a mapping is made between the channel identifier to an empty queue, since sent messages are not queued. The timers also suffer a small change, since now an unique name is associated to every timed call. Thus, the timers, $\mathcal{T}$, are now a map of strings into the representation of a timed call, $(l(\vec{v}) \times \textsf{Int} \times \textsf{Int})$. The representation of a timed call has now only two integers (the periodicity of the call and the next invocation) due to the removal of the expire parameter in the timers. The last modification to the syntactic categories of the virtual machine was related to messages. With the restructuring of the communication semantics, the output queue is not used, and therefore now it is only represented by an empty queue, ϵ. These messages on the input queue have the same format as before, the only difference now is that we have multiple queues, one for every input channel opened. Since now the receive operation has a channel associated, a lookup for the channel is made in the connection manager, which returns the associated queue.

The initial state of the virtual machine is obtained by loading the top-level module, at offset 0 of program, p, containing the function init, used to initiate the computation. Thus, loading a program $\mathcal{P}$ is done by function boot() obtaining the following result:

$$\mathcal{P}, \langle 0, \mathcal{M}_0, \{\}, (0, \epsilon, \epsilon, \langle \textsf{loadc } 0, \textsf{call2}, \textsf{return} \rangle, \langle \text{"init"} \rangle), \epsilon, \{\} \rangle \leftarrow \textsf{boot}(\mathcal{P})$$

There are only small changes in the initial state, where now we start the program by calling the function init. Also, the input and output queues were removed from the machine state, and instead we have the connection manager, $\mathcal{N}$, as the last item, that is initially empty.

In terms of state transitions, Table 5.1 shows the instructions affected by the changes, and also the transitions associated to the three new instructions created. The open command is used to open a communication channel, and expects a string, v, on top of the operand stack. This string contains the link to the channel and it is used to open the respective channel. When a channel is created a new queue is associated to it. The information about the channel is stored in the connection manager, $\mathcal{N}$, for further use. The instruction close also expects a string, v, on top of the operand stack, using it to access the representation of the channel, closing it, and removing its information from the connection manager.

Continuing on the instructions for communication, both send and receive were obviously affected by the introduction of channels in the language. In the case of the send, it now expects on top of the operand stack, not only the function name and its arguments, but also a string, v, identifying the channel through which the message will be sent. In order to execute the instruction the channel must be opened and it must also be an output channel. This is checked with isSource. If these conditions are satisfied the message is created and sent immediately. The receive instruction, needs also the link to the channel, v, in order to fetch from the associated input queue, $\mathcal{Q}$, the first message received from that specific channel. Like in the instruction send, it checks if the channel is open and is an input channel with isSink, and if it is the case, the virtual machine fetches its input queue. Both instructions in the case of the channel not being open or not matching the sensor type (sink or source), crash the virtual machine and the execution halts.

The timed calls were restructured and therefore it affected the behavior of timers in the virtual machine. The instruction timer needs to return a unique label, containing the name of the function and a timestamp, and for that reason it is now present a string, v, which represents that unique label and it is placed on top of the operand stack by this instruction, in order to be returned afterwards. The string v, is also used to store the timers, since now they are represented by a map from a string to the representation of a timer (BoundedTimedTask). In order to terminate a timer, the instruction kill was added. It expects a string, v, on top of the operand stack, containing the name of the respective timer. Then, a lookup for the specific timer is made and it is terminated and removed from the mapping.

The updated virtual machine architecture is presented on Figure 5.6. The sender thread was removed and whenever a message is sent the main thread is responsible for the packing of the function call into a message bytecode and for the dispatch of the message. Also, we have now multiple receiver threads which correspond to the

Table 5.1: Updated transition rules for communication and timed call management instructions used by the virtual machine

$\mathcal{B}[i]$	Assumptions	Transitions
open	$v \notin \mathcal{N}$	$t \to t'$
		$i \to i+1$
		$\mathcal{S} : v \to \mathcal{S}$
		$\mathcal{N} \to \mathcal{N} \cup (v, \epsilon)$
close		$t \to t'$
		$i \to i+1$
		$\mathcal{S} : v \to \mathcal{S}$
		$\mathcal{N} \uplus (v, \_) \to \mathcal{N}$
send	$\mathcal{M}_0(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$	$i \to i+1$
	$j_1 = \lvert\vec{v}_1\rvert$	$\mathcal{S} : \vec{v}_1 : l : v \to \mathcal{S}$
	$\mathsf{isSource}(v)$	
receive	$\mathsf{isSink}(v)$	$t \to t'$
	$\mathcal{Q} = \mathcal{N}(v)$	$i \to i+1$
		$\mathcal{S} : v \to \mathcal{S}$
		$\mathcal{Q} :: \langle l, \vec{v} \rangle \to \mathcal{Q}$ (incoming queue)
		$\mathcal{R} \to l(\vec{v}) :: \mathcal{R}$ (run queue)
timer	$\mathcal{M}_0(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$	$i \to i+1$
	$j_1 = \lvert\vec{v}_1\rvert$	$\mathcal{S} : j : \vec{v}_1 : l \to \mathcal{S} : v$
	$v = \mathsf{newId}(l, t)$	$\mathcal{T} \to \mathcal{T} \cup \{(v, (l(\vec{v}_1), j, t+j))\}$ (timers)
kill		$t \to t'$
		$i \to i+1$
		$\mathcal{S} : v \to \mathcal{S}$
		$\mathcal{T} \uplus \{(v, \_)\} \to \mathcal{T}$
(interrupt)	$t = t'$	$\mathcal{T} \uplus \{(v, (l(\vec{v}), j, t'))\} \to \mathcal{T} \cup \{(v, (l(\vec{v}), j, t'+j))\}$
		$\mathcal{R} \to l(\vec{v}) :: \mathcal{R}$

different input channels opened in the application. These receiver threads contain their own input queue from where the main thread fetches the function calls. A final addition to the architecture was the timer threads. Timed calls are now processed

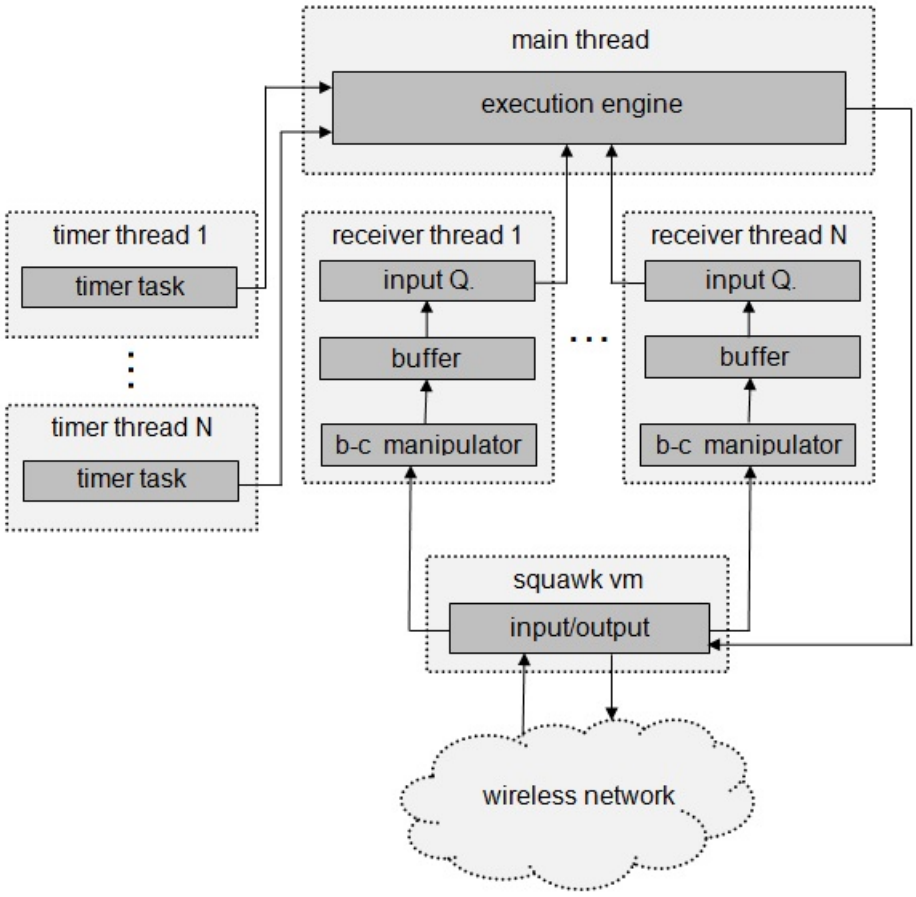

Figure 5.6: Updated architecture of the Callas virtual machine

by independent threads that put the respective function call, whenever is time, for executing in the main thread.

In this chapter we described the impact of the changes in the language. Using this updated version of the Callas programming language a large demo was created to verify its usability. This demo will be presented in the next chapter.

Chapter 6

Exponential Smoothing Demo

With the new syntax and semantics we decided to test its usability by the creation of a more complex application that will be presented in this section. We will start by presenting the problem, and then explaining how it was implemented in Callas. This demo is based on the application described in [2]. The idea of this application is to use a predictive model in order to reduce the communication between sensors in an application where a physical quantity (in this case the light value), is monitored at periodic, fixed intervals.

6.1 Adaptive Model Selection

The main objective in Adaptive Model Selection (AMS) [3] is the reduction of the communication on wireless sensor networks performing periodic data collection tasks, by using predictive models that approximate the real measurements.

In some applications, the exact sensor measurements are not required, only an approximation of the value is enough. This is the idea of AMS, where models are used to approximate the measurements collected by a wireless sensor by means of time series prediction techniques. A model refers to a parametric function that predicts, at time $t + m$, $m \in \mathbb{N}$, the measurement of the sensor. Formally, the model is a function

$$\begin{aligned} h_\theta :& \mathrm{X} \to \mathbb{R} \\ & x \mapsto \hat{x}[t + m] = h_\theta(x) \end{aligned}$$

where $x \in X$ is the input of the model, which is normally a vector of measurements, θ

is a vector with the parameters of the model, and $\hat{x}[t + m]$ is the approximation of the model h to the measurement $x[t + m]$.

In a normal periodic sensor application, the measurements are sent to the basestation at a certain rate, but in AMS the parameters of the model are sent instead. The basic technique works as follows, a threshold value, e, is defined that represents the error tolerance of the application. The sensor node locally assess if the prediction $\hat{x}[t + m]$ made by the model is within $\pm e$ of the true measurement $x[t + m]$. Whenever the difference between the real measurement and the prediction is larger than the threshold value, a new set of parameters is sent to the basestation. If the difference is not significant no communication is needed, and the basestation assumes that the prediction is correct. The definition of this threshold value obviously depends on the application requirements, but the higher the value, the less communication occurs.

6.2 Exponential Smoothing

Exponential smoothing is a time series prediction technique that has different proposed variants, and has been shown to perform well in a large number of time series [10]. The simpler version is the simple exponential smoothing, which consists in a weighted average, s, of the past measurements. The weighted average is calculated as

$$s[t] = \alpha x[t - 1] + (1 - \alpha)s[t - 1]$$

where $0 \leq \alpha \leq 1$ represents the data smoothing factor. In this case the predictions are given with $\hat{x}[t + m] = s[t]$.

However, in order to achieve better approximations to the real measurements, it is normally used the double exponential smoothing. This model takes into account the trend of the data and in order to obtain the predictions we need to compute

$$\begin{aligned} s[t] &= \alpha x[t - 1] + (1 - \alpha)(s[t - 1] + b[t - 1]) \\ b[t] &= \beta(s[t] - s[t - 1]) + (1 - \beta)b[t - 1] \end{aligned}$$

where $0 \leq \beta \leq 1$ represents the trend smoothing factor. The predictions are given with $\hat{x}[t + m] = s[t] + mb[t]$.

Note that the computational cost of the exponential smoothing is very low, and therefore makes it an ideal model for implementation in resource-constrained wireless

sensors. Another advantage of the utilization of predictive models is the possibility to compress data, since the number of model parameters sent to the basestation is normally less than the sensor values.

6.3 Demo Implementation

The physical quantity being monitored in this application is the light intensity. The application implemented in Callas uses the double exponential smoothing model to predict the light values and therefore save unnecessary communication. Thus, it is required the specification of the values for the data and trend smoothing factors, α and β. These values are used to calculate the model parameters, $s[t]$ and $b[t]$. In order to present the data from the Callas application we use an interface implemented in Java. This interface gets the values from the Callas application output and builds a graph with it. Figure 6.1 shows a set of four snapshots of the demo where the readings were exactly the same for all. Readings were taken every 5 seconds for 15 minutes. We have the real sensor values without using a prediction model on the top left graph. The Model 1 (top right) ignores the trend ($\beta = 0$) and uses $\alpha = 1$. The Model 2 (bottom left) has $\alpha = 0.8$ and $\beta = 0.2$, and finally the Model 3 (bottom right) uses $\alpha = 0.4$ and $\beta = 0.6$. This means that in Model 3 we are giving more importance to the trend of the data than the real sensor readings when calculating the prediction value, $\hat{x}[t + m]$. On the other hand, in Model 2 the trend of the data is also taken in account but has less impact in the prediction, $\hat{x}[t + m]$, than the real values.

The error tolerance for this application was set as $e = 10lux$, since the readings were taken in an office where the light intensity normally varies between 10 and 125 lux. As it is possible to see the α and β values affect directly the model. For example, Model 1, basically mantains the same light value until an update is made. The graph is drawn in two colors, where color red represents the times where the model was updated in the basestation. For Models 1 and 2, the saved transmissions were about 70%, but in Model 3 this value dropped to 64%. The saved transmission are computed every t dividing the number of times that the prediction model did not need any update, by t.

The demo here presented in detail computes a single prediction model that uses the double exponential model. Also, the demo is created to work in a network with only one sensor and the basestation. It could be possible to increase the number of sensors in the application. The basestation needed to maintain information about the prediction

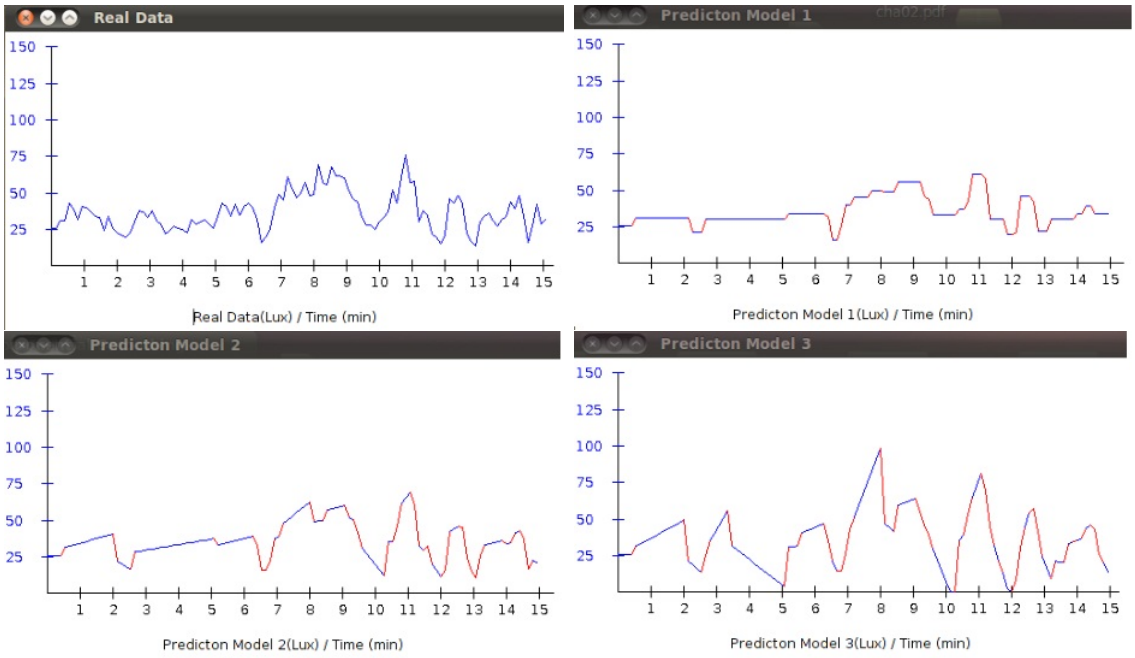

Figure 6.1: Snapshot of the Double Exponential Demo. From top to bottom, left to right: Real Data, Model 1 ($\alpha = 1$, $\beta = 0$), Model 2 ($\alpha = 0.8$, $\beta = 0.2$), Model 3 ($\alpha = 0.4$, $\beta = 0.6$)

models for every sensor in the network, and calculate the respective models every iteration. Figure 6.2 shows the application type for this demo, with several modules defined. The module Channels is used to store the links for the communication. There is a module Timers to store any needed information about a timer, in order to be able to terminate it later in the execution. The FixedParameters module contains the information about the fixed parameters values needed for the prediction model. The model parameters that are sent from the nodes to the basestation are stored in the module ModelParameters. In order for the nodes to keep track of the model parameters still in use by the basestation, a module CurrentParameters is used to store these values. The StoredData is used to maintain information about time t, the previous readings of the node in order to calculate $s[t]$, and also the number of saved transmissions, savedTrans. In order for the basestation to know if the model was updated, a boolean updated is held in the module Updated, along with the information about value m, used to calculate $\hat{x}[t+m]$. Finally the main module Sensor extends all other modules, and contains the methods that implement the vital code for the application.

In our implementation, the sensor node waits for a message from the basestation before starting to collect data. When the message is received by the node, the

```
6  defmodule Timers:
7    string timerID()
8
9  defmodule FixedParameters:
10   double alfa()
11   double beta()
12   double errorTolerance()
13
14 defmodule ModelParameters:
15   double st()
16   double bt()
17
18 defmodule CurrentParameters:
19   double currSt()
20   double currBt()
21
22 defmodule StoredData:
23   double xt()
24   double t()
25   double savedTrans()
26
27 defmodule Updated:
28   bool updated()
29   double m()
30
31 defmodule Channels:
32   string input()
33   string output()
34
35 defmodule Sensor(StoredData, Channels, ModelParameters, Timers, Updated,
36  FixedParameters, CurrentParameters):
37   Nil init()
38   Nil firstData()
39   Nil predict()
40   Nil updateData(ModelParameters newParameters)
41   Nil printData(double prediction, double t, bool updated,
42    double savedTrans)
43   Nil listen(string c2)
```

Figure 6.2: Exponential Smoothing Demo: iface.caltype

first light reading is made, and the model parameters are initialized and sent to the basestation. After this initialization process, both node and sink run the prediction

model every 5 seconds, and in the case of the difference between the prediction and the real measurement is larger than the error tolerance, the node sends the updated model parameters to the basestation. If that is not the case, then no communication occurs, and the basestation assumes that the prediction is correct. The application also keeps track of the percentage of saved transmissions that is updated every t. The complete source code for this application is provided with this thesis in Appendix A.

Chapter 7

Conclusions

In this thesis we presented a series of experiments with the Callas programming language and its virtual machine with the general goal of assessing its usability, expressiveness, and to optimize the virtual machine in terms of resource footprint. New derived constructs were implemented to facilitate the programming of common patterns in the Callas language. The program syntax was changed, so a top-level module is always present in the applications. This makes the syntax of the programs cleaner while allowing the initial state of the virtual machine to be typable, a problem detected in the previous specification.

The timed calls were completely restructured. Callas now uses distinct threads to run this type of calls. This modification allowed a cleaner implementation and opens different possibilities in terms of energy management. Now, the main thread sleeps until being awaked by a timer, and the threads responsible for the timed calls only awake whenever it is time to place a call on the run queue, sleeping in between. The changes also included major work in the communication semantics of the language. This was a limitation in the language that was partially overcome, allowing now the programmer to use user-defined channels. Previously, nodes only sent messages to the basestation, and the basestation always sent messages to all nodes. Now it is possible for the user to choose the appropriate channel for the communication, including creating distinct communication groups in the sensor network. For the purpose of this thesis, only the radiogram protocol for the communication was supported.

With all the changes applied to the Callas language a large demo was implemented in the new model. This application uses a prediction model to avoid excessive communication between the nodes and the sink. It also shows that it is possible to create

more elaborated applications for wireless sensor networks in Callas.

As for future work, the goal is to adapt the virtual machine to support other type of services, such as HTTP and RMI. This will enable a growth of the language potential, since it opens new possibilities for applications. For example, it would be possible to process data outside the sensor network. Since the sensor devices normally have small processing and storing capacity, this could be useful in applications that use complex algorithms to analyze data.

Appendix A

Double Exponential Smoothing Demo

The application in Callas contains the following files:

- main.calnet
- sunspot.caltype
- iface.caltype
- sink.callas
- node.callas

```
44 #File main.calnet
45
46 interface = iface.caltype
47 externs   = sunspot.caltype
48
49 sensor:  code = sink.callas
50 sensor:  code = node.callas
```

```
51 #File sunspot.caltype
52
53 defmodule Extern:
54     bool   setLEDColor(long pos, long red, long green, long blue)
55     bool   setLEDOn(long pos, bool isOn)
56     bool   logLong(long val)
57     bool   logDouble(double val)
58     bool   logString(string val)
59     string macAddr()
60     long   getTime()
61     long   battLevel()
62     long   getLuminosity()
63     double getTemperature()
64     double getAccelX()
65     double getAccelY()
66     double getAccelZ()
67     double getAccel()
68     double getInclX()
69     double getInclY()
70     double getInclZ()
71     double longToDouble(long val)
```

```
72  #File iface.caltype
73  defmodule Nil:
74    pass
75
76  defmodule Timers:
77    string timerID()
78
79  defmodule FixedParameters:
80    double alfa()
81    double beta()
82    double errorTolerance()
83
84  defmodule ModelParameters:
85    double st()
86    double bt()
87
88  defmodule CurrentParameters:
89    double currSt()
90    double currBt()
91
92  defmodule StoredData:
93    double xt()
94    double t()
95    double savedTrans()
96
97  defmodule Updated:
98    bool updated()
99    double m()
100
101 defmodule Channels:
102   string input()
103   string output()
104
105 defmodule Sensor(StoredData, Channels, ModelParameters, Timers, Updated,
106  FixedParameters, CurrentParameters):
107   Nil init()
108   Nil firstData()
109   Nil predict()
110   Nil updateData(ModelParameters newParameters)
111   Nil printData(double prediction, double t, bool updated,
112    double savedTrans)
113   Nil listen(string c2)
```

```
114 #File sink.callas
115 from iface import *
116
117 module main of Sensor:
118   def init(self):
119     mem = load
120     #opening communication channels
121     c1 = mem.output()
122     c2 = mem.input()
123     open c1
124     open c2
125     #ask for first data
126     listen(c2) every 5000
127     predict() every 5000
128     select c1 send firstData()
129
130   def updateData(self, newParameters):
131     install newParameters
132     module update of Updated:
133       def updated(self): True
134       def m(self): 1.0
135     install update
136     pass
137
138   def predict(self):
139     mem = load
140     alfa = mem.alfa()
141     beta = mem.beta()
142     errorTolerance = mem.errorTolerance()
143     #DOUBLE EXPONENTIAL SMOOTHING
144     #getting the values from the last iteration
145     st = mem.st()
146     bt = mem.bt()
147     t = mem.t()
148     m = mem.m()
149     savedTrans = mem.savedTrans()
150     updated = mem.updated()
151     if updated:
152       savedTrans = savedTrans
153     else:
154       savedTrans = savedTrans +. 1.0
155     prediction = st +. m *. bt
156     negative = prediction <. 0.0
157     if negative:
```

```
158         prediction = 0.0
159       else:
160         prediction = prediction
161       module newData of StoredData:
162         def xt(self): prediction
163         def t(self): t +. 1.0
164         def savedTrans(self): savedTrans
165       mem ||= newData
166       module update of Updated:
167         def updated(self): False
168         def m(self): m +. 1.0
169       mem ||= update
170       store mem
171       mem.printData(prediction,t,updated,savedTrans)
172
173     def printData(self, prediction, t, updated, savedTrans):
174       extern logString("PData\n")
175       extern logDouble(t)
176       extern logString("\n")
177       extern logDouble(prediction)
178       extern logString("\n")
179       percentage = savedTrans / t
180       extern logDouble(percentage)
181       extern logString("%\n")
182       if updated:
183         extern logString("true\n")
184       else:
185         extern logString("false\n")
186       pass
187
188     def firstData(self):
189       pass
190
191     def alfa(self):
192       0.2
193
194     def beta(self):
195       0.6
196
197     def errorTolerance(self):
198       5.0
199
200     def xt(self):
201       0.0
202
```

```
203   def st(self):
204     0.0
205
206   def bt(self):
207     0.0
208
209   def currSt(self):
210     0.0
211
212   def currBt(self):
213     0.0
214
215   def t(self):
216     0.0
217
218   def updated(self):
219     False
220
221   def savedTrans(self):
222     0.0
223
224   def m(self):
225     1.0
226
227   def timerID(self):
228     "";
229
230   def input(self):
231     "radiogram://:90";
232
233   def output(self):
234     "radiogram://broadcast:90";
235
236   def listen(self, c2):
237     select c2 receive
```

```
238 #File node.callas
239 from iface import *
240
241 module main of Sensor:
242   def init(self):
243     mem = load
244     #opening communication channels
245     c1 = mem.output()
246     c2 = mem.input()
247     open c1
248     open c2
249     #timer to receive first message
250     timerID = listen(c2) every 200
251     module newTimer of Timers:
252       def timerID(self): timerID
253     mem ||= newTimer
254     store mem
255     pass
256
257   def firstData(self):
258     extern setLEDColor(0, 0, 255, 0)
259     extern setLEDOn(0, True)
260     mem = load
261     timerID = mem.timerID()
262     kill timerID
263     #first data
264     longLux = extern getLuminosity()
265     lux = extern longToDouble(longLux)
266     module updatedData of StoredData:
267       def xt(self): lux
268       def t(self): 1.0
269       def savedTrans(self): 0.0
270     mem ||= updatedData
271     module newParameters of ModelParameters:
272       def st(self): lux
273       def bt(self): 0.0
274     mem ||= newParameters
275     module newCurrParam of CurrentParameters:
276         def currSt(self): lux
277         def currBt(self): 0.0
278     mem ||= newCurrParam
279     store mem
280     predict() every 5000
281     c1 = mem.output()
```

```
282       select c1 send updateData(newParameters)
283
284     def predict(self):
285       mem = load
286       alfa = mem.alfa()
287       beta = mem.beta()
288       errorTolerance = mem.errorTolerance()
289       #DOUBLE EXPONENTIAL SMOOTHING
290       longLux = extern getLuminosity()
291       lux = extern longToDouble(longLux)
292       #getting the values from the last iteration
293       oldxt = mem.xt()
294       oldst = mem.st()
295       oldbt = mem.bt()
296       t = mem.t()
297       currSt = mem.currSt()
298       currBt = mem.currBt()
299       m = mem.m()
300       savedTrans = mem.savedTrans()
301       #calculating s[t]
302       expr1 = 1.0 -. alfa
303       expr2 = oldst +. oldbt
304       st = alfa *. oldxt +. expr1 *. expr2
305       #calculating b[t]
306       expr1 = st -. oldst
307       expr2 = 1.0 -. beta
308       bt = beta *. expr1 +. expr2 *. oldbt
309       #AMS
310       diff = 0.0
311       prediction = currSt +. m *.currBt
312       if prediction >. lux:
313         diff = prediction -. lux
314       else:
315         diff = lux -. prediction
316       needsUpdate = diff >. errorTolerance or st <. 0.0
317       if needsUpdate:
318         module newData of StoredData:
319           def xt(self): lux
320           def t(self): t +. 1.0
321           def savedTrans(self): savedTrans
322         mem ||= newData
323         module newParameters of ModelParameters:
324           def st(self): st
325           def bt(self): bt
326         mem ||= newParameters
```

```
327         module newCurrParam of CurrentParameters:
328           def currSt(self): st
329           def currBt(self): bt
330         mem ||= newCurrParam
331         module newUpdated of Updated:
332           def updated(self): True
333           def m(self): 1.0
334         mem ||= newUpdated
335         c1 = mem.output()
336         store mem
337         select c1 send updateData(newParameters)
338       else:
339         module newData of StoredData:
340           def xt(self): prediction
341           def t(self): t +. 1.0
342           def savedTrans(self): savedTrans +. 1.0
343         mem ||= newData
344         module newParameters of ModelParameters:
345           def st(self): st
346           def bt(self): bt
347         mem ||= newParameters
348         module updated of Updated:
349           def updated(self): False
350           def m(self): m +. 1.0
351         c1 = mem.output()
352         store mem
353         pass
354
355     def updateData(self, newData):
356       pass
357
358     def printData(self, st, t, updated, savedTrans):
359       pass
360
361     def alfa(self):
362       0.2
363
364     def beta(self):
365       0.6
366
367     def errorTolerance(self):
368       5.0
369
370     def xt(self):
371       0.0
```

```
372
373   def st(self):
374     0.0
375
376   def bt(self):
377     0.0
378
379   def currSt(self):
380     0.0
381
382   def currBt(self):
383     0.0
384
385   def t(self):
386     0.0
387
388   def updated(self):
389     False
390
391   def m(self):
392     1.0
393
394   def savedTrans(self):
395     0.0
396
397   def timerID(self):
398     "";
399
400   def input(self):
401     "radiogram://:90";
402
403   def output(self):
404     "radiogram://broadcast:90";
405
406   def listen(self, c2):
407     select c2 receive
```

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] Y. Le Borgne and G. Bontempi. Time series prediction for energy-efficient wireless sensors: Applications to environmental monitoring and video games. In *Proceedings of the S-Cube*, 2012.

[3] Y. Le Borgne, S. Santini, and G. Bontempi. Adaptive model selection for time series prediction in wireless sensor networks. *Signal Processing*, 87(12):3010 – 3020, 2007.

[4] A. Boulis, C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services (MobiSys'03)*, pages 187–200. ACM Press, 2003.

[5] T. Cogumbreiro, P. Gomes, F. Martins, and L. Lopes. Safe-By-Design Programming Languages for Wireless Sensor Networks. Technical Report TR DCC-2011-09, Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto, June 2011. Available at `http://www.dcc.fc.up.pt/dcc/Pubs/TReports/`.

[6] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *EmNets'04 Workshop*, 2004.

[7] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *In Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks*, 2005.

[8] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *ICDCS'05*, pages 653–662. IEEE Press, 2005.

[9] W. F. Fung, D. Sun, and J. Gehrke. COUGAR: The network is the database. In *SIGMOD'02*. ACM Press, 2002.

[10] E. S. Gardner. Exponential smoothing: The state of the art. *Journal of Forecasting*, 4(1):1–28, 1985.

[11] 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 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11, 2003.

[12] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *In DCOSS*, pages 126–140. Springer, 2005.

[13] J. Lifton, D. Seetharam, M. Broxton, and J. Paradiso. Pushpin Computing System Overview: a Platform for Distributed, Embedded, Ubiquitous Sensor Networks. In *Proceedings of the Pervasive Computing Conference (Pervasive'02)*. Springer-Verlag, 2002.

[14] L. Lopes, F. Martins, and J. Barros. *Middleware for Network Eccentric and Mobile Applications*, chapter 2, pages 25–41. Springer-Verlag, 2009.

[15] L. Lopes, F. Martins, M. S. Silva, and J. Barros. A Process Calculus Approach to Sensor Network Programming. In *International Conference on Sensor Technologies and Applications (SENSORCOMM'07)*, pages 451–456. IEEE Computer Society, 2007.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 2005.

[17] F. Martins, L. Lopes, and J. Barros. Towards Safe Programming of Wireless Sensor Networks. *Electronic Proceedings in Theoretical Computer Science*, 17:49–62, 2010.

[18] 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.

[19] 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 the ACM International Conference on Virtual Execution Environments (VEE'06)*, June 2006.

[20] SunSPOT. Project Sun SPOT, 2004. `http://www.sunspotworld.com/`.

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

[22] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, NSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[23] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52:2292–2330, August 2008.