Journal of Systems Architecture 63 (2016) 16–32

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A safe-by-design programming language for wireless sensor networks

Luís Lopes [a,*], Francisco Martins [b]

[a] CRACS/INESC-TEC & Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
[b] LASIGE & Faculdade de Ciências, Universidade de Lisboa, Campo Grande, 1749-016 Lisboa, Portugal

ARTICLE INFO

Article history:
Received 29 November 2013
Revised 17 January 2016
Accepted 18 January 2016
Available online 28 January 2016

Keywords:
Programming language
Compiler
Virtual machine
Type safety
Wireless sensor network

ABSTRACT

Wireless sensor networks are notoriously difficult to program and debug. This fact not only stems from the nature of the hardware, but also from the current approaches for developing programming languages and runtime systems for these platforms. In particular, current systems do not place enough stress on providing formal descriptions of the language and its runtime system, and on proving static properties, like type-safety and soundness. In this paper, we present the design, specification, and implementation of a programming language and a runtime system for wireless sensor networks that are safe by design. We say this in the sense that we can statically detect a large set of would-be runtime errors, and that the runtime system will not incorrectly execute an application, once the latter is deployed. We have a full prototype implementation of the system that supports SunSPOT devices, the simulation tool VisualSense, and local computer networks for fast deployment and testing of applications. Development is supported by an IDE implemented on top of the Eclipse tool that embeds both the compiler and the virtual machine seamlessly, and is used to produce software releases.

1. Introduction

Wireless sensor networks (WSN) are one of the most challenging hardware platforms to program. They are gatherings of large numbers of small physical devices, commonly referred to as sensors or motes, capable of sensing the environment. The communication infra-structure is based on low-power wireless technologies and uses ad-hoc networking protocols [1]. The difficulty in programming WSN results from the unique characteristics of these platforms, especially when compared with other ad-hoc networks such as MANETs. The sensor devices are extremely limited in terms of hardware resources, namely CPU and memory, and energy, typically provided by batteries. Their deployment at remote locations makes physical access to the devices, e.g., for maintenance and debugging, in many cases difficult if not impossible, or simply not practical.

There are many proposals for programming languages for WSN providing the programmers with distinct levels of hardware and network awareness and distinct programming abstractions [2]. Given the aforementioned restrictions, programming languages for wireless sensor networks are often tightly coupled with the underlying operating system, which is typically very lightweight and modular [3–7]. At the very lowest level of programming, running on the bare hardware, we have languages such as Pushpin [8], and languages such as TinyScript and Mottle that use a thin abstraction layer for the hardware provided by a virtual machine [9,10]. Abstracting away from the hardware there are languages like the (ubiquitous) component-based language nesC [11] tightly coupled with its host operating system TinyOS [7]. Higher up in the abstraction level we find macroprogramming languages that allow programmers to abstract away, not only from devices, but also from the network infra-structure, by resorting to sophisticated compilers to automate code generation and deployment. They provide abstractions such as: streams, e.g., Regiment [12]; databases, e.g., TinyDB [13] and Cougar [14]; regions, e.g., Abstract Regions [15]; agents, e.g., Sensorware [16] and Agilla [17]; web-services, e.g., IrisNet [18].

Despite the diversity of proposals, applications for wireless sensor networks are difficult to debug and often produce runtime errors. The problem stems from the fact that most languages are built in a fairly ad-hoc way, typically by first identifying a set of adequate programming abstractions and implementing a compiler that maps the high-level syntax directly into native code or, more commonly, into an intermediate language representation, nesC code for example, or some form of bytecode. Macroprogramming languages are illustrative of this state of affairs. Regiment [12], for example, a strongly typed functional macroprogramming language, is compiled into a low-level token

* Corresponding author. Tel.: +351 960376714.
E-mail addresses: lblopes@dcc.fc.up.pt (L. Lopes), fmartins@di.fc.ul.pt (F. Martins).

http://dx.doi.org/10.1016/j.sysarc.2016.01.004
1383-7621/

machine language, which is then itself compiled into a nesC implementation of the runtime based on the distributed token machine model. The complex compilation scheme makes it rather difficult to establish a link between the semantics of the language and that of the corresponding runtime system, especially in the absence of a formal specification for the programming language and for the runtime system.

Runtime errors in sensor network applications can have multiple origins: (Type I) device malfunction or interference from the environment; (Type II) semantic errors in the application; (Type III) the runtime system does not preserve the language semantics; (Type IV) the compiler generates code that does not preserve the language semantics.

Errors of Type I are difficult or impossible to eliminate in most deployments. Type II errors can be controlled by imposing an adequate programming discipline, e.g., enforced by a type system, and by carefully testing the application before deployment. Type III and IV errors are far more subtle but very important, as they may undermine a deployment with seemingly unexplainable errors and result in significant extra costs. Type III errors can be eliminated by proving that the specification of the runtime preserves the semantics of the source language. This of course still leaves some margin for errors in the programming of the runtime, but these can be weeded out through conventional tests. Finally, Type IV errors can be eliminated by proving that the compiler generates code that preserves the semantics of the original program. This is usually called a certified compiler.

In short, errors of types II to IV can be eliminated by providing a formal specification for the programming language semantics and for the runtime semantics, and proving static properties that relate them, e.g., type-safety and soundness. Language type-safety ensures that well-typed programs do not give rise to runtime protocol errors. A compiler for a type-safe programming language can statically type-check code and identify would-be runtime protocol errors, before the application is deployed over the network. *This is possible since the full application, including the code to be run at the sink(s) and the code to be run at the nodes, is compiled as a unit, allowing for communication protocol errors to be prematurely detected.* This addresses errors of Type II. On the other hand, the soundness property ensures that the underlying runtime system preserves the semantics of the programming language. This is achieved by implementing the runtime system based on an abstract specification (e.g., a virtual machine) that can be proved to preserve the semantics of the programming language. This addresses errors of Type III. We do not address Type IV errors in this paper. This is the subject of current research.

To illustrate the design and implementation principles that we propose, we present the step by step development of Callas [19], a programming language for WSN. The language and its semantics are specified using a formal model, based on concurrency theory [20,21]. The runtime system for the language was specified in the form of a virtual machine, defined as a state transition system. Elsewhere we proven that the language is type-safe and that therefore well-typed programs do never produce a large set of runtime errors [22]. Moreover, we also proved that the runtime system preserves the semantics of the language, a property also known as soundness, and thus correctly executes Callas applications. In this paper we overview the design of the programming language and of the runtime system, and describe a full prototype implementation of this framework. The prototype includes a language compiler, a modular virtual machine that supports multiple hardware and software platforms, e.g., SunSPOT networks [23] and the VisualSense simulator [24] for deployment, and a development environment based on an Eclipse plugin that seamlessly embeds both the compiler and the runtime system and is used for software releases.

To our knowledge the use of process calculi to model and design languages for sensor networks is a novel approach. Previous work on process calculi for wireless systems is scarce and focuses on communication protocols. Prasad [25] established the first process calculus approach to modeling broadcast based systems. Later work by Ostrovský et al. [26] established the basis for a higher-order calculus for broadcasting systems. More recently, Mezzetti and Sangiorgi [27] discuss the use of process calculi to model wireless systems, again focusing on the details of the lower layers of the protocol stack (e.g., collision avoidance) and by establishing an operational semantics for the networks.

In the recent past the Internet of Things (IoT) gained a lot of attention both from the Academia and from the Industry. The IoT is a network of physical objects or "things" embedded with electronics, software, sensors, and network connectivity, which enables these objects to collect and exchange data. This paper focus on a more restricted scenario, that of wireless sensor networks. WSN aggregates a myriad of devices with similar hardware and software characteristics that autonomously collect, eventually process, and send data to gateways. From the IoT perspective, the data emanating from the gateway would be thought of as a single resource. IoT poses interesting challenges of its own, like, for instance, the interoperability between things. On the other hand, programming WSN is by itself difficult, error-prone, and correcting bugs can be difficult if not impossible after deployment in the field. It is this last problem that we tackle in this paper and propose a solution that involves a language that is demonstrably correct, thus significantly diminishing the sources of error for WSN applications.

The remainder of the paper is structured as follows. Section 2 presents the Callas language: its syntax, semantics, and briefly overviews the language safety results. Section 3 presents the Callas virtual machine: the bytecode format, the reduction rules, and briefly describes the soundness result. Sections 4 and 5 describe the prototype implementation that includes: the language compiler, the virtual machine with support for several hardware and software platforms, and a development environment based on the Eclipse tool [28]. Section 6 describes related work on programming languages and virtual machines for wireless sensor networks. Finally, Section 7 ends the paper with some conclusions and perspectives for future work.

2. The programming model

This section aims at describing Callas, a programming language for sensor networks that offers constructs to describe local computations, communications, code mobility, and code updates. The language is based on a calculus [19,22] with the goal of establishing a foundation for developing programming languages and runtime systems for sensor networks.

We start by presenting the language with a running example to illustrate the programming style of Callas (Section 2.1). Thereafter, we introduce an abstract core language (Section 2.2) suitable for defining its formal semantics (Section 2.3), of which we present only an excerpt to emphasize the foundations of Callas. In Section 2.4 we state informally a type safety result—the interested reader may refer to [22] for the details.

2.1. The Callas programming language

We introduce the Callas language by example, programming a device that periodically reads the ambient's temperature and sends it over the network, as presented in Listing 1. A Callas program is a sequence of *type declarations* followed by a code *module* that implements the type for the devices in a WSN, known as `Device`. Other declared modules arise as submodules of this top-level module. We adopt Python's line-oriented syntax, where

Listing 1. A program for periodically transmitting the sensed temperature to the network.

```
1  # type declarations
2  defmodule Nil:
3    pass
4
5  defmodule Sampler:
6    Nil sample()
7
8  defmodule Device(Sampler):
9    Nil init()
10
11 # terms: declare module Node, init contains the intial code to run
12 module node of Device:
13   def init(self):
14     sample() every 60*10
15   def sample(self):
16     curTemp = sys.getTemp()
17     send log(curTemp)
```

indentation (the number of spaces in the beginning of a line) demarcates syntactic terms.

The program starts with three type declarations (lines 2–3, 5–6, and 8–9). The first type declaration begins with the keyword `defmodule`, followed by a type identifier `Nil` (must be capitalized) that binds and introduces the declared module type. The body of a module type is a sequence of *function signatures*, which declare the type of the result, the function name, and the types of the parameters. In line 3, we define an empty module type (without functions). Keyword **pass** defines an empty sequence of syntactic terms, used to declare an empty syntactic block. In lines 5–6, we find the declaration of a type `Sampler`, a module with a function named `sample` that expects no arguments and returns empty modules. Finally, in lines 8–9, we declare the type for the device, `Device`, that extends that of `Sampler`. Thus, any implementation of `Device` must provide an implementation for its own `init` function, and also for a `sample` function.

The program ends with the definition of the `Device` module, lines 12–17. The first line holds the *module header* and then the *module body* follows. The module header begins with the keyword `module`, succeeded by a variable `node` that binds the module—variables must begin with a lower-case letter—, followed by the keyword `of`, then the type identifier `Device` that specifies the type of the declared module, and terminates with a colon (`:`).

Similarly to a module declaration, a function declaration comprises a *function header* and a *function body*. The header (e.g., in line 13) starts with the keyword `def`, followed by the name of the function `init`, and by one or more (comma-separated) parameters in parenthesis. The first parameter in any function is the module itself, denoted as **self**, e.g., to allow for recursive calls. The function body is a sequence of terms. Functions are second-class values, meaning that they cannot be handled directly, e.g., passed as an argument of another function. Note that, as in Python, when a line ends with a colon the remaining lines are a syntactic group with an increased indentation. Function `init` is the first function to be executed in a Callas program and must always be defined, with the given signature, in the `Device` module. In this case, the body of function `init` simply starts a timer that invokes the function `sample` every 10 min (the time unit is the second). This allows the device to perform periodic tasks and moreover conserve energy between samplings. The body of function `sample` consists of two terms. The first assigns to variable `curTemp` the result from an operating system call that gets the ambient temperature from a sensor in the device. The second term is a network call to function `log`, passing the value of variable `curTemp` as an argument. Expression **send** is an asynchronous function call to neighboring devices. This expression yields as a result an empty module (that is propagated as the outcome of function `sample`). There are no guaranties that any device in the network picks up these remote function calls. The programmer must develop protocols for making sure messages are delivered or to recover in case messages are lost. The syntax of values comprises two categories: built-in values and variables. We adopt the Python's syntax for built-in values as well as for unary and binary operations.

To conclude our first example, a network of devices executing the code in Listing 1 needs one or more devices that are programmed to receive the data and process it. Devices responsible for collecting the data generated by a WSN are usually called *sinks*. Listing 2 presents the Callas code for recording the maximum received temperature in the memory of the sink. Line 8 declares type `Device` by extending the type `MaxTemp`. Line 9 defines the signature of the function `init`, followed, in line 10, by the signature of a function `log` with a typed parameter named `temp` of type **float**. Finally, line 11, defines a function that will be responsible for handling incoming data from the devices. Each typed parameter consists of a type and a name (the latter used for documentation proposes only). Types are any of the built-in types—the integer type `int`, the float type `float`, and the Boolean type `bool`—and the module types, given by (capitalized) type identifiers. After the type declarations, we have the definition of the `Device` module, here associated with an instance variable `sink`, lines 14–30. Function `init` programs a timer that executes a function `listenForData` every 10 min. Function `log` expects two parameters: the module itself and the temperature `temp`. The function loads the previous known maximum temperature by invoking function `maxTemp`. Then, in case `temp` is greater than `maxTemp`, the function builds a new module, bound to variable `newMax` (lines 21–23), and updates the received temperature by storing module `newMax` in the devices's memory, which updates function `maxTemp` to hold the new maximum sensed value (lines 24–26). Note that when the else branch is omitted, its value is the empty module.

Some additional detail is required here. The memory of a device contains a code module that may be updated dynamically, by the application, throughout the lifetime of the device. For accessing the memory of a device we use expressions **load** and **store**. In line 24, we load the code of the device and save it in variable `m`, assign to variable `m` a new module, by composing the modules in variables `m` and `newMax` (line 25), and store the new module in memory (line 26). Expression $x||y$ merges the functions of both x and y into a new module, giving preference to the functions of module y in case of name clashes (i.e., the same function signature

Listing 2. A program for storing the maximum temperature received.

```
1  # type declarations
2  defmodule Nil:
3    pass
4
5  defmodule MaxTemp:
6    float maxTemp()
7
8  defmodule Device(MaxTemp):
9    Nil init()
10   Nil log(float temp)
11   Nil listenForData()
12
13 # terms: declare module sink, init contains the code to be executed at startup
14 module sink of Device:
15   def init(self):
16     listenForData() every 60*10
17   def log(self, temp):
18     maxTemp = self.maxTemp()
19     needsUpdate = temp > maxTemp
20     if needsUpdate:
21       module newMax of MaxTemp:
22         def maxTemp(self):
23           temp
24       m = load
25       m = m || newMax      # update function maxTemp
26       store m
27   def listenForData(self):
28     receive
29   def maxTemp(self):
30     0                      # initial minimum temperature is 0
```

appearing on both modules). It allows for dynamic code update as long as the types of both modules are compatible. The syntax for operator || is based on the asymmetric merge operator of the record calculus [29].

Function `listenForData` (lines 27–28) checks the device's incoming queue for messages, reads one, if available, and returns the empty module. A message that is read is eventually executed by the device by calling the appropriate function in the module stored in its memory. Finally, function `maxTemp` sets the initial value for the maximum temperature, in the present case zero (lines 29–30).

Note that, although we use the terms *function* and *asynchronous function call*, the Callas programming model is conceptually similar to *event-based* programming models. In fact, asynchronous function calls and timed calls $l(\vec{v})$ can be seen as asynchronous events where function name l is the event identifier and also the name of the *call-back*. Thus, sending a message $l(\vec{v})$ is like generating an event in some network neighborhood. A receiving device captures the event and calls the corresponding call-back l that must reside in its memory.

This simple example can be made more sophisticated by introducing the possibility of dynamically patching the code in the devices from within the application, avoiding major network bootstraps. This capability is interesting both for allowing the evolution of applications without disrupting their execution, and for instrumenting the code for online debugging. The example is supported by the fact that in Callas modules are first class values, that is, they may be passed as arguments to calls and in particular they may be propagated by messages in the network.

The implementation of the new code for the devices is depicted in Listing 3. The main difference lies in the definition of a new module, `Patcher`, that defines two functions, `patch` and `listenForPatches`. As the names imply, function `patch` replaces a function currently in the device by another that it receives from the network. Function `listenForPatches` periodically checks the network for patching messages. The implementation of the `Device` module is very similar to the first version. Function `init`, lines 15–17, starts an extra timer that listens for patches. The implementation given for function `sample`, lines 18–20, is the same as in Listing 1, but may be replaced at any time if a patching message arrives. Function `patch`, lines 21–24, receives a patching message with a `Sampler` module as an argument and uses it to replace the current implementation of that sub-module in the device (in this case only function `sample` is affected). Finally, function `listenForPatches`, lines 25–26, just listens for patching messages from the network. Listing 4 displays the corresponding code for the sink, adapted from Listing 2. The code starts with the declaration of the type `Patcher` and of the type `Device` that extends both `Sampler` and `Patcher`. Henceforth the main differences occur in the implementation of functions `init` and of functions `patch` and `listenforPatches`. Function `init`, lines 22–28, first builds a new `Sampler` module that captures the temperature from an hypothetical second temperature sensor in the devices, lines 23–26. This could be interesting, e.g., if some sort of hardware calibration or debugging was taking place. Another option would be to implement the `sample` function in such a way that it returns a synthetic stream of temperature values that may be used to diagnose problems in the application or in the hardware. This `Sampler` module is then sent to the device using an asynchronous message in line 27. As usual, the sink sets up a listener for data, line 28.

The complete (concrete) grammar for Callas is depicted in Fig. 1. We use boldface fonts to identify terminal symbols. Notation $\vec{x}$ denotes a (possibly empty) sequence of elements of syntactic category x.

2.2. Abstract syntax

To define the semantics of our language we need to distil the concrete syntax into a core language containing the necessary

Listing 3. A program for sending the sensed temperature over to the network periodically.

```
1  defmodule Nil:
2    pass
3
4  defmodule Sampler:
5    Nil sample()
6
7  defmodule Patcher:
8    Nil patch(Sampler sampler)
9    Nil listenForPatches()
10
11 defmodule Device(Sampler, Patcher):
12   Nil init()
13
14 module node of Device:
15   def init(self):
16       sample() every 60*10
17       listenForPatches() every 60*10
18   def sample(self):
19     curTemp = sys.getTemp()
20     send log(curTemp)
21   def patch(self, sampler):
22     m = load
23     m = m || sampler   # update sampler function
24     store m
25   def listenForPatches(self):
26     receive
```

constructs to express Callas and that abstracts away from concrete details (e.g., end of lines, spaces, blocks).

Fig. 2 describes this *core Callas*. We retain just three syntactic categories: expressions, e, modules, m, and values, v. In what concerns expressions, we add **let** and **if** constructs. The **let** expression handles the binding constructs uniformly (variable assignment and module definition), makes the scope of the bindings explicit, and enforces an evaluation order on expressions. Recall that in the concrete syntax of Callas, assignments and module definitions introduce new variables that are visible until the end of the current block, and that a block is defined by the indentation level of lines. The let construct, **let** $x = e_1$ **in** e_2, first evaluates expression e_1, binds its result to the new variable x, whose scope is e_2, and then uses this value when evaluating expression e_2. The conditional **if** e_1 **then** e_2 **else** e_3 that evaluates condition e_2 when expression e_1 is true, and evaluates expression e_3 otherwise. Modules, m, are collections of functions, as before, but the new syntax allows for explicitly treating modules as values, and the module construct **module** x **of** T : ¶ $\vec{f}$ is expressed as a **let** for binding the module with variable x in a given scope.

In Fig. 3 we formalize the translation rules from concrete to core Callas syntax. We skip module type declarations when translating programs. The translation of a conditional at the head of a sequence of terms $\vec{t}_3$ is directly mapped into a conditional expression; we compose the new conditional with the translation of each branch, and use a **let** to enforce sequential execution of the conditional and then of the continuation (the translation of $\vec{t}_3$). Note that variable x plays no role in the continuation expression. A module **module** x **of** T : ¶ $\vec{f}$ at the head of a sequence of terms $\vec{t}$ becomes a binding of module $\{[[\vec{f}]]\}$ to variable x in the scope of the translation of $[[\vec{t}]]$; the new module results from the translations of functions $\vec{f}$; assignment $x = e$ ¶ at the head of a sequence of terms $\vec{t}$ is represented as a binding of expression e to x in the continuation $[[\vec{t}]]$. Note that expressions are not translated at all. If a term is the last in the program, then there is no need to introduce a new binding, and therefore the term is represented just as its value. For instance, assignment $x = e$ ¶ is translated as e (the result of the assignment term). Applying function $[[\cdot]]$ to the program listed in Fig. 1 we obtain its core representation in Fig. 5.

2.3. Semantics

The runtime environment for Callas is presented in Fig. 4 and focuses on the devices and on the network. Sensor networks, S, are concurrent compositions of devices, represented as terms of the form $[e, R \triangleright m, T]_t^{I,O}$, and of the empty network, denoted by ϵ. Each device is composed of an expression e being evaluated, a queue of pending expressions R, a module m with the installed functions, a set of timers T for periodically calling functions in the installed code, queues for incoming/outgoing messages from/to the network (I/O), and the current time t. Messages are passivated function calls denoted as $\langle l(\vec{v})\rangle$, sometimes abbreviated as q, and are the moving entities in the network.

The meaning of programs in Callas is defined using an operational semantics, in particular a reduction system combined with a structural congruence relation (omitted) defined over S. Structural congruence identifies programs that are considered syntactically equivalent even when their textual representation is different. It allows semantically equivalent networks to be rewritten so that the reduction rules may be applied. We give an excerpt of the reduction relation rules in Figs. 5 and 6 that, in our opinion, is sufficient for giving a flavor of how reduction is defined. The interested reader should refer to [22] for the complete specification of the reduction relation. In general the rules have the form

$$\frac{assumption_1 \quad \ldots \quad assumption_n}{S \rightarrow S'}$$

that is read as: network S evolves in one reduction step to network S' (runs in one step) provided that $assumption_1, \ldots, assumption_n$ are satisfied. In order to control the evaluation order, we only allow reduction to happen within the **let** construct. To simplify the notation we write $\langle e' \rangle$ as an abbreviation for **let** $x = e'$ **in** e, where variable x and expression e are arbitrary, but are chosen to be the same in the context of a reduction rule. Alternatively, we could present the semantics using evaluation contexts, as we did in [22], but the current approach is more amenable to be used as the specification for a run-time system. The reduction steps are controlled by an internal clock t. The time for the next activation of every programmed timed call is checked against the current clock time

Listing 4. A program for storing the maximum temperature received.

```
# type declarations
defmodule Nil:
  pass

defmodule MaxTemp:
  float maxTemp()

defmodule Sampler:
  Nil sample()

defmodule Patcher:
  Nil patch(Sampler sampler)
  Nil listenForPatches()

defmodule Device(MaxTemp, Sampler, Patcher):
  Nil init()
  Nil log(float temp)
  Nil listenForData()

# terms: declare module sink, init contains the code to be executed at startup
module sink of Device:
  def init(self):
    module dbgSampler of Sampler:
        def sample(self):
          # using hypothetical secondary temperature sensor
          curTemp = sys.getTemp2()
          send log(curTemp)
    send patch(dbgSampler)
    listenForData() every 60*10
  def log(self, temp):
    maxTemp = self.maxTemp()
    needsUpdate = temp > maxTemp
    if needsUpdate:
      module newMax of MaxTemp:
          def maxTemp(self):
            temp
      m = load
      m = m || newMax          # update function maxTemp
      store m
  def listenForData(self):
    receive
  def maxTemp(self):
    0                          # initial minimum temperature is 0
```

Listing 5. The abstract syntax of the sampling node.

```
{
 init   = (self)
   sample() every 60*10
 sample = (self)
   let curTemp = sys.getTemp() in send log(curTemp)
}
```

using the predicate *noEvent*. Reduction is driven by running expression e, which executes the associated action and advances the clock. We assume that each instruction consumes an unspecified number of processor cycles and in most of the rules the clock moves forward from some t to some t'.

Rule (1) makes a synchronous call to a function l provided by the underlying operating system or a library, and immediately receives a value v. These calls are typically used to access the hardware of the device, e.g., to read sensors or to activate actuators. Rule (2) handles the interaction between a device and the network by packing a call $l(\vec{v})$ into a message $\langle l(\vec{v})\rangle$ and placing it in the outgoing queue, O. The **receive** operation (Rule (3)) takes a message $\langle l(\vec{v})\rangle$ from the incoming queue, I, unpacks it, and appends the corresponding call to function $x.l(\vec{v})$ to the device's run queue, $R :: \textbf{let } x = \textbf{load in } x.l(\vec{v})$. The code for l is installed in the device in m and will be loaded when the function is selected for execution, as the expression inserted in the run-queue implies.

Rule (4) handles calls to functions in modules. It selects the code for the function, $m_2(l) = (\textbf{self } \vec{x})e$, replaces the parameters by the arguments, passing the current module m_2 as the first argument in variable **self**, and runs the resulting expression, $e[m_2\ \vec{v}/\textbf{self } \vec{x}]$.

Rule (5) allows modules to be merged, with functions in the left-hand module, m_1, being replaced with functions of equal name in the right-hand module, m_2. For example, given $m_1 = \{l_1 = (\vec{x_1})P_1\ l_2 = (\vec{x_2})P_2\}$ and $m_2 = \{l_2 = (\vec{x_2})P_2'\}$, the result of $m_1 || m_2$ is the module $m_3 = \{l_1 = (\vec{x_1})P_1\ l_2 = (\vec{x_2})P_2'\}$. This rule is pivotal since

p	$::= \vec{d}\ m$	*Programs*
d	$::=$ **defmodule** $T : \P\ \vec{s}$	*Type Decl.*
s	$::=$	*Func. Sigs.*
	$\tau\ l(\tau\vec{x})\ \P$	signature
	| **pass** ¶	empty sequence
τ	$::=$	*Types*
	bool | **int** | **float**	basic types
	| T	module type
t	$::=$	*Terms*
	pass ¶	inaction
	| $x = e\ \P$	assign
	| m	module impl.
	| $e\ \P$	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
	| $v \mid\mid v$	merge modules
	| $v.l(\vec{v})$	function call
	| $l(\vec{v})$ **every** v	timed call
	| **send** $l(\vec{v})$	communication
	| **receive**	communication
v	$::=$	*Values*
	x	variable
	| **sys**	system module
	| ... | 0 | ...	integer
	| **True** | **False**	boolean
	| ... | 0.0 | ...	floating point

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

Fig. 1. The syntax of Callas.

e	$::=$	*Expressions*
	...	same as e in Figure 1
	| **let** $x = e$ **in** e	sequence
	| **if** e **then** e **else** e	conditional
m	$::= \{l_i = (\vec{x}_i)e_i\}_{i\in I}$	*Modules*
v	$::=$	*Values*
	...	same as v in Figure 1
	| m	modules

Fig. 2. The syntax for core Callas.

$$
\begin{aligned}
[\![\vec{d}\ \vec{t}]\!] &= [\![\vec{t}]\!] \\
[\![\textbf{if}\ e : \P\ \vec{t}_1\ \textbf{else}\ : \P\ \vec{t}_2\ \vec{t}_3]\!] &= \textbf{let}\ x = \textbf{if}\ e\ \textbf{then}\ [\![\vec{t}_1]\!]\ \textbf{else}\ [\![\vec{t}_2]\!]\ \textbf{in}\ [\![\vec{t}_3]\!], x \text{ is fresh} \\
[\![\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}\ \vec{t}]\!] &= \textbf{let}\ x = \{[\![\vec{f}]\!]\}\ \textbf{in}\ [\![\vec{t}]\!] \\
[\![x = e\ \P\ \vec{t}]\!] &= \textbf{let}\ x = e\ \textbf{in}\ [\![\vec{t}]\!] \\
[\![e\ \P\ \vec{t}]\!] &= \textbf{let}\ x = e\ \textbf{in}\ [\![\vec{t}]\!], x \text{ is fresh} \\
[\![\textbf{if}\ e : \P\ \vec{t}_1\ \textbf{else}\ : \P\ \vec{t}_2]\!] &= \textbf{if}\ e\ \textbf{then}\ [\![\vec{t}_1]\!]\ \textbf{else}\ [\![\vec{t}_2]\!] \\
[\![\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}]\!] &= \{[\![\vec{f}]\!]\} \\
[\![x = e\ \P]\!] &= e \\
[\![e]\!] &= e \\
[\![\textbf{def}\ f(\vec{x}) : \P\ \vec{t}]\!] &= f = (\vec{x})[\![\vec{t}]\!] \\
[\![\epsilon]\!] = [\![\textbf{pass}]\!] &= \{\}
\end{aligned}
$$

Fig. 3. Abstraction rules.

S	$::=$	*Sensors*
	ϵ	empty network
	| $S \mid S$	composition
	| $[e, R \triangleright m, T]_t^{I,O}$	sensor
R	$::= e_1 :: \cdots :: e_n$	run-queue
T	$::= \{(l_i(\vec{v}_i), v_i, v_i)\}_{i\in I}$	timed calls
q	$::= \langle l(\vec{v})\rangle$	messages
I, O	$::= q_1 :: \cdots :: q_n$	message queues

Fig. 4. The syntax of Callas runtime environment.

it allows the dynamic reprogramming of a device without violating the type of the running application.

Rule (6) programs a timer for a call to a function l installed in the device, i.e., whose code is in m. Note that T' includes the information of the new timer, namely, the information to call the triggered function, $l(\vec{v})$, the time interval between calls, v, and the next point in time that the trigger should be fired, $t + v$. The notation $A \uplus B$, for sets A and B, means $A \cup B$ for $A \cap B = \emptyset$. When predicate *noEvent* evaluates to false, rule (7) comes into action, placing a timed function call $l(\vec{v})$ in the run-queue. The execution of the call is delegated to rule (4). Note that Rule (7) needs to trigger all the calls to the timers that are due, hence it does not advance the clock (t remains the same after the reduction step).

The reduction semantics for networks (Fig. 6) is orthogonal to that for in-device processing. Communication occurs by broadcasting messages over wireless channels to devices in the neighborhood of the broadcasting device. Rule (8) handles the distribution of outgoing messages by delivering such messages to the incoming queues of receiving devices. The semantics abstracts away from the lower level protocol layers that are responsible for message delivery, i.e., the networking, MAC, and physical layers. In other words, it simply assumes that a message from a sending device *may* eventually reach the target devices. A broadcast starts with the formation of an initially empty membrane, denoted by $\{\epsilon\}$ (rule omitted). Multiple applications of the rule for broadcast (Rule 8) then distribute a message to the destination devices. The rule says that a sending device, with a message q in its outgoing queue O_1, and with a subnetwork S of neighboring devices within this membrane, eventually transfers q to the incoming queue, I_2, of another device, which is then absorbed by the membrane. The broadcast ends with the dissolution of the membrane created by the sending device (rule omitted). It is important to note that: (a) the *membrane* is not a physical entity, rather it is a formal construct that guarantees that a device never receives the same message twice *directly* from the sending device; and (b) devices within the membrane are not allowed to proceed with reduction,

$$\frac{\text{noEvent}(T,t) \qquad v = sysCall(l,\vec{v})}{[\langle \textbf{sys}\ .l(\vec{v})\rangle, R \triangleright m, T]_t^{I,O} \rightarrow [\langle v\rangle, R \triangleright m, T]_{t'}^{I,O}} \tag{1}$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textbf{send}\ \ l(\vec{v})\rangle, R \triangleright m, T]_t^{I,O} \rightarrow [\langle \{\}\rangle, R \triangleright m, T]_{t'}^{I,O::\langle l(\vec{v})\rangle}} \tag{2}$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textbf{receive}\,\rangle, R \triangleright m, T]_t^{\langle l(\vec{v})\rangle::I,O} \rightarrow [\langle \{\}\rangle, R :: \textbf{let}\ \ x = \textbf{load}\ \textbf{in}\ \ x.l(\vec{v}) \triangleright m, T]_{t'}^{I,O}} \tag{3}$$

$$\frac{m_2(l) = (\textbf{self}\ \ \vec{x})e \quad \text{noEvent}(T,t)}{[\langle m_2.l(\vec{v})\rangle, R \triangleright m_1, T]_t^{I,O} \rightarrow [\langle e[m_2\ \vec{v}/\textbf{self}\ \ \vec{x}]\rangle, R \triangleright m_1, T]_{t'}^{I,O}} \tag{4}$$

$$\frac{\text{noEvent}(T,t) \qquad m_3 = merge(m_1,m_2)}{[\langle m_1 \mid\mid m_2\rangle, R \triangleright m, T]_t^{I,O} \rightarrow [\langle m_3\rangle, R \triangleright m, T]_{t'}^{I,O}} \tag{5}$$

$$\frac{T' = T \cup \{(l(\vec{v}), v, t+v)\} \qquad \text{noEvent}(T,t)}{[\langle l(\vec{v})\ \textbf{every}\ \ v\rangle, R \triangleright m, T]_t^{I,O} \rightarrow [\langle \{\}\rangle, R \triangleright m, T']_{t'}^{I,O}} \tag{6}$$

$$\frac{t' \leq t \qquad T' = T \cup \{(l(\vec{v}), v, t+v)\}}{[e, R \triangleright m, T \uplus \{(l(\vec{v}), v, t')\}]_t^{I,O} \rightarrow [e, \textbf{let}\ \ x = \textbf{load}\ \textbf{in}\ \ x.l(\vec{v}) :: R \triangleright m, T']_t^{I,O}} \tag{7}$$

Fig. 5. Reduction semantics for devices (some rules omitted).

$$[\ \_\ ]_{t_1}^{I_1,q::O_1}\{S\} \mid [\ \_\ ]_{t_2}^{I_2,O_2} \rightarrow [\ \_\ ]_{t_1}^{I_1,q::O_1}\{S \mid [\ \_\ ]_{t_2}^{I_2::q,O_2}\} \tag{8}$$

Fig. 6. Reduction semantics for sensor networks (some rules omitted).

meaning that, from the point of view of these devices, communication is instantaneous.

2.4. Type safety and absence of runtime errors

The static semantics of the Callas programming language is provided in the form of a type system [22], whose details we omit in the current paper. In the sequel we describe the formal results that we have proved for the type system and reduction rules of the operational semantics. The first result, *subject reduction*, states that types are invariant under reduction, i.e., the type of a program does not change as it executes. Therefore, if a program is well typed (denoted by sequent $\Gamma \vdash S$) and it performs a computational step ($S \rightarrow S'$), the remaining program to execute (S') must also be well typed ($\Gamma \vdash S'$). Formally, this intuition is captured by the following theorem.

Theorem 1 (Subject reduction). *If $\Gamma \vdash S$ and $S \rightarrow S'$, then $\Gamma \vdash S'$.*

We also proved the *type safety* of the language, meaning that well-typed programs do not produce a class of runtime errors, namely: (a) any given function call is always made in a module that contains that function and that such a call matches the function's signature; (b) the same validations as in (a) apply to messages traveling the network, in the sense that whenever a device receives a message from the network it has a local function (in the device's memory) that can be correctly called; and (c) updating a module always preserves its type, that is, the signatures of the functions it contains. *Since this verification is done statically by a compiler, with the full code for the application available - both sink and node sides, it is possible to detect prematurely communication protocol errors. These applications will not be type-checked successfully.* We write $S \overset{\text{err}}{\longmapsto}$ to denote networks that are "stuck" due to one of the problems described above, and write $S \overset{err}{\not\longmapsto}$ for $\neg(S \overset{\text{err}}{\longmapsto})$.

The type safety result ensures that well-typed networks never get "stuck" and is stated as follows:

Theorem 2 (Type safety). *If $\Gamma \vdash S$, then $S \overset{err}{\not\longmapsto}$.*

A corollary of this result is the *absence of runtime errors* of the kinds described. In other words, well-typed networks do not produce runtime errors at any time during the execution of a program. Therefore, if a program is well typed (denoted by sequent $\Gamma \vdash S$) and if, after performing an arbitrary number of reductions ($\rightarrow^*$), we get a network S' that is never "stuck" ($S' \overset{err}{\not\longmapsto}$).

Corollary 3 (Absence of runtime errors). *If $\Gamma \vdash S$ and $S \rightarrow^* S'$, then $S' \overset{err}{\not\longmapsto}$.*

These results show that an appropriate type discipline imposed on Callas applications allows us to ensure statically, at compile time, that a class of common runtime errors will never arise, in this case errors of Type II.

3. The Callas virtual machine

To execute Callas applications we wanted a runtime system that provides an abstraction for the hardware platforms. We designed a custom bytecode format and a virtual machine specification that allowed the execution of Callas applications. A runtime representation of the bytecode may be seen in Fig. 7.

$$
\begin{array}{ll}
\textit{program} & \mathcal{P} \in \textsc{ArrayOf}(\mathcal{D}) \\
\textit{module bytecode} & \mathcal{D} \in \textsc{MapOf}(\textbf{String} \mapsto \mathcal{F}) \\
\textit{function bytecode} & \mathcal{F} \in \textbf{Int} \times \textbf{Int} \times \textbf{Int} \times \mathcal{B} \times \mathcal{U} \\
\textit{values} & \mathcal{U} \in \textsc{ArrayOf}(v) \\
\textit{function code} & \mathcal{B} \in \textsc{ArrayOf}(c) \\
\textit{value} & v \in \textbf{Bool} \cup \textbf{Int} \cup \textbf{Float} \cup \mathcal{M} \\
\textit{module} & \mathcal{M} \in \textsc{MapOf}(\textbf{String} \mapsto \mathcal{F} \times \textsc{ArrayOf}(v)) \\
\textit{instruction} & c \in \{\mathsf{loadm}\ i, \mathsf{loadm2}\ i, \mathsf{call}, \mathsf{call2}, \mathsf{merge}, \mathsf{send}, \mathsf{receive}, \\
 & \quad \mathsf{timer}, \mathsf{return}, \mathsf{jmp}\ i, \mathsf{ift}\ i, \mathsf{loadb}, \mathsf{storeb}, \mathsf{loadc}\ i, \mathsf{load}\ i, \\
 & \quad \mathsf{store}\ i, \mathsf{add}, \mathsf{sub}, \ldots\}
\end{array}
$$

Fig. 7. The byte-code format.

A bytecode program, $\mathcal{P}$, is composed of an array of module definitions. A module definition, $\mathcal{D}$, in the array is a map from strings (the function names) onto tuples representing the functions. A function, $\mathcal{F}$, is represented by a tuple that contains: tree integers that hold the number of parameters, of free variables, and of local variables for the function; a bytecode array $\mathcal{B}$ that holds the code for the function; and an array $\mathcal{U}$ that contains constants present in the source code. Every byte-code array for a function ends with a return statement. The virtual machine manipulates values that can be basic data types (Booleans, integers, and floats) and modules, $\mathcal{M}$. The latter are dynamic instances of module definitions created by constructing a closure, which involves collecting the free variables for each of the functions in the module definition, in the given execution context, and storing them together with the module's byte code. The instruction-set includes instructions for manipulating modules, making calls, moving data, network I/O, control-flow and basic arithmetic, and logic operations. The virtual machine is stack-based and thus expects operands at the top of the operand stack. Load and store instructions are used to move values between the environment frame and the operand stack. Constant values are loaded onto the operand stack with special load instructions. This allows for a simple and compact instruction set with few addressing modes.

3.1. The virtual machine data-structures

The state of the virtual machine is given by the pair $\mathcal{P}, \langle \textbf{Int}, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R} \rangle^{\mathcal{I}}_{\mathcal{O}}$. Besides the byte-code for the program, $\mathcal{P}$, the elements of the state proper have the following informal meaning, in order:

- **Int** an integer value representing an internal clock for the machine;
- $\mathcal{M}$ is a module containing the functions that have been installed in a sensor. The module can be updated during the execution of the device. Functions cannot be added or be removed, they can just be replaced by already existing ones;
- $\mathcal{T}$ is a set of programmed timed function calls. Each timed-call is composed of an operand-stack, holding the environment for the call, and two integers, which represent the period of the call and the time of the next call , respectively;
- $\mathcal{C}$ is a call-stack. Each component, a call-frame, is 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.
- $\mathcal{R}$ is a queue of pending function calls $l(\vec{v})$. These calls are loaded onto the call-stack when the previous call returns;
- $\mathcal{I}/\mathcal{O}$ are input-output queues of passivated function calls. These are used by the virtual machine to interact with the lower layers of the sensor network protocol stack.

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

Fig. 8. The syntactic categories of the virtual machine.

The items in arrays, stacks, and queues of a syntactic category α are written $\langle \alpha_1, \ldots, \alpha_n \rangle$, $\alpha_1 : \cdots : \alpha_n$ and $\alpha_1 :: \cdots :: \alpha_n$, respectively. Empty arrays, stacks, and queues are denoted ϵ. A summary of the components of the virtual machine is given in Fig. 8.

3.2. The initial state

Every program $\mathcal{P}$ has an entry point, the function that is first executed when the program starts. This function is called *init* and is part of the top level module of the program, at offset 0. The function has no parameters and no free variables. The module does not have free variables also, since it is the top level module in the source program and there are no global variables. Thus, the initial state of the virtual machine is obtained by loading the program $\mathcal{P}$. This operation is performed by a function `boot()` with the following result:

$$
\begin{array}{l}
\mathcal{P}, \langle 0, \mathcal{M}_0, \{\}, (0, \epsilon, \epsilon, \langle \texttt{loadc}\ 0, \texttt{call2}, \texttt{return} \rangle, \langle \text{''init''} \rangle), \epsilon \rangle^{\epsilon}_{\epsilon} \\
\quad \leftarrow \texttt{boot}(\mathcal{P})
\end{array}
$$

The function builds a closure, $\mathcal{M}_0$, for the top level module, loads it into the virtual machine, and installs a short sequence of byte-code that starts the program by calling *init*. The byte-code loads the identifier for the function from the constant array using instruction `loadc 0` and then calls the function, using `call2`. The final `return` instruction ends the byte-code sequence and clears the frame from the call-stack. Naturally, in the initial state, the incoming-, outgoing-, and run-queues are empty. The set of timed calls is also empty. Henceforth, the execution of the program proceeds through a series of state transitions designed to match the operational semantics given in Section 2.

3.3. Reduction rules

The reduction rules describe the state transitions of the virtual machine as the bytecode instructions are executed. An assortment of these rules is presented in Table 1 to give the reader a grasp of the complete specification. It is used not only to guide the implementation of the prototype, but also to allow proving its

Table 1
An assortment of transition rules for the virtual machine.

$\mathcal{B}[i]$	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)
	$j_1 = \lvert\vec{v}_1\rvert,\ j_3 = \lvert\vec{0}\rvert$	$\to \mathcal{C} : (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : (0, \mathcal{E}', \epsilon, \mathcal{B}', \mathcal{U}')$	
merge	$\mathcal{M}_3 = \mathsf{merge}(\mathcal{M}_1, \mathcal{M}_2)$	$t \to t'$	
		$i \to i+1$	
		$\mathcal{S} : \mathcal{M}_2 : \mathcal{M}_1 \to \mathcal{S} : \mathcal{M}_3$	
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	$\mathcal{M}_0(l) = (\mathcal{F}, \vec{v}_2)$	$t \to t'$	
	$\mathcal{F} = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$	$i \to i+1$	
	$j_1 = \lvert\vec{v}_1\rvert$	$\mathcal{S} : j : \vec{v}_1 : l \to \mathcal{S}$	
		$\mathcal{T} \to \mathcal{T} \cup \{(l(\vec{v}_1), j, t+j)\}$	(timers)
(interrupt)	$t = t'$	$\mathcal{T} \uplus \{(l(\vec{v}), j, t')\} \to \mathcal{T} \cup \{(l(\vec{v}), j, t'+j)\}$	
		$\mathcal{R} \to l(\vec{v}) :: \mathcal{R}$	

soundness with respect to the operational semantics of the language. Where appropriate, $I = \{0 \ldots n\}$, is a set of consecutive integer indexes. Before making a call to a function we must first load a copy of the corresponding module on to the stack. **Loading** the jth module in the bytecode for a program involves some preparatory work. First, the map containing the bytecode for the module is collected in $\mathcal{P}[j] = \{l_k \mapsto \mathcal{F}_k\}_{k\in I}$, where l_k is the function name and $\mathcal{F}_k = (j_1, j_2, j_3, \mathcal{B}, \mathcal{U})$ is a tuple that contains information about function l_k, namely, the number of parameters (j_1), free variables (j_2), local variables (j_3), and its bytecode and constant arrays. The case for which none of the functions has free variables is handled by the instruction loadm and results in a particularly simple closure for the module. The general case is handled by instruction loadm2 and differs in that the resulting module includes arrays of values, captured from the current environment, that are the values for the free variables for each of the module's functions. These values are placed at the top of the operand stack, listed per function, $\vec{v}_k$ representing the values of the free variables for the function named l_k. The execution of loadm2 results in a module, $\{l_k \mapsto (\mathcal{F}_k, \vec{v}_k)\}_{k\in I}$, that is left on top of the operand stack. **System Calls**, handled by the call instruction, allow programs to interact with the basic hardware functionality (true or simulated), namely to access sensors and actuators. The function name, l, and the arguments to the call, $\vec{v}$, are placed on top of the operand stack and consumed. The call is handled by a built-in function of the virtual machine that interfaces with the underlying operating system or with a library. The virtual machine has internal information on the arity of the function l and uses it to prepare the call to $\mathsf{sysCall}(l, \vec{v})$. The returned value is placed on the top of the operand stack. **Calling** a function in a module is one of the most fundamental operations of the virtual machine. The relevant instruction, call2, expects the name of the function, l, the module it belongs, $\mathcal{M}$, and the arguments to the call, $\vec{v}_1$, to be placed on top of the operand stack. The instruction consumes these values and, while doing so, it collects runtime information on $\mathcal{M}$ and l: the function's bytecode, $\mathcal{B}'$, the function's constants, $\mathcal{U}'$, the values for the free variables, $\vec{v}_2$, and the size of the environment frame, $j_1 + j_2 + j_3$. A new environment frame, $\mathcal{E}$, is built with the arguments to the call, $\mathcal{M}\vec{v}_1$, the values of the free variables, $\vec{v}_2$, and extra space for the local variables, $\vec{0}$, with size j_3. The instruction builds a new call-frame with the aforementioned information and pushes it on top of the call-stack, $\mathcal{C}$. As would be expected, the new frame has 0 instruction counter and empty, ϵ, operand stack. **Merging** modules is another important trait of the Callas programming language. As we have described, it allows for the dynamic update of code modules in a disciplined way, i.e., the update ($\mathcal{M}_2$) must respect the type of the module to update ($\mathcal{M}_1$). The instruction, merge, expects both modules at the top of the stack and produces a new module that merges both and is left at the top of the stack.

Sending a message over a wireless channel involves building it from information obtained from the top of the operand stack. The instruction send inspects the definition of function l in the module $\mathcal{M}_0$ to find out its arity, and thus the number of values to be fetched from the stack. In a well-typed program, this arity will coincide with that of whatever function l that exists in other devices, although the implementations may differ. The function name, l, and the call arguments, $\vec{v}$, are fetched and a message is built, $\langle l, \vec{v}\rangle$, that is then added to the end of the output queue, $\mathcal{O}$, to be processed. **Receiving** a message involves checking the input queue, $\mathcal{I}$. The instruction receive takes the message at the head of the queue and transfers it to the end of the run-queue where it waits for execution as a pending call. All the information required to build the pending call is contained in the message. If $\mathcal{I}$ is empty the instruction returns immediately without side-effects, other than the updates of the program counter and of the system time. **Periodic Tasks** are first programmed using the instruction timer. The programming of such a task involves storing the call, $l(\vec{v}_1)$, the period, j, and the next invocation time, $t + j$, in a tuple in $\mathcal{T}$. As above, to find out the number of values to extract from the stack, $\lvert\vec{v}_1\rvert$, the instruction inspects $\mathcal{M}_0$, more specifically, the entry for

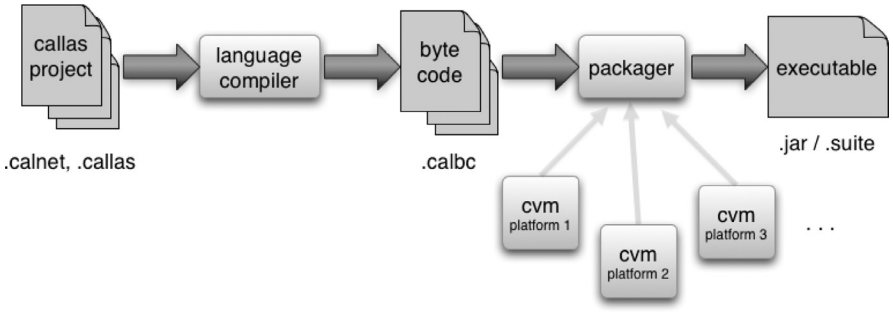

Fig. 9. The compilation process.

function l. **Triggering Periodic Tasks** at a given instant t' evolves an interrupt-like mechanism: the execution halts, a pending call, $l(\vec{v})$ is placed at the end of the run queue, and the execution resumes. The timer is updated for the next invocation at $t' + j$. This process does not involve the execution of an instruction and any progress in time, since more than one timer may be triggered at the same time instance.

3.4. Soundness of the virtual machine

We have proved that the specification of the CVM given in this section preserves the semantics of Callas programs. More precisely, we proved that, assuming a compilation function C as defined in [30], if we start with a bytecode program $C(e)$ that is the result of the compilation of a Callas expression e, and that this program, as it is executed by the virtual machine, evolves through a sequence of transitions into another expression $C(e')$, which also corresponds to the compilation of a Callas expression e', then there exists a one step reduction using the Callas semantics from e to e'. Formally, this result could be stated as follows:

Theorem 4 (Runtime Soundness). *If* $C(e) \rightarrow {}^{*}C(e')$, *then* $e \rightarrow e'$.

This theorem establishes a deep link between the operational semantics of the Callas programming language and the operational semantics for the runtime. It implies that bytecode programs generated by the Callas compiler (and thus free of runtime protocol errors) will be executed by the runtime system in accordance with the operational semantics for the language as given in Section 2.

4. Prototype implementation and deployment

As a proof-of-concept we have built a Software Development Kit (SDK) for programming WSN with Callas that supports three different platforms, one for real life SunSPOT devices, on for the VisualSense simulator, and one that runs on personal computers over LANs (Callas UDP). Since we abstract the devices with the CVM, the same compiler is used for all three platforms. Each platform, however, must provide its own implementation of the CVM.

The compilation process is composed of two steps (Fig. 9). In the first step, the source code for each file is transformed into a corresponding Callas bytecode file. The second step embeds the bytecode files in a .jar or .suite file, as appropriate for the target platform (see below), at a specific point within the package hierarchy. This bundle is then deployed in the target platform and executed. When the application starts to execute, e.g., `startApp()` for SunSPOT MIDlets, the virtual machine loads the embedded bytecode into runtime data structures (see below), which are used thereafter. To give some flavor of the size of the source (Java), the code generation component consists of eight classes that represent 1400 lines of code, half of these represent test cases.

All versions of the CVM target a Java runtime system (JVM). SunSPOT applications run on top of the Squawk [31] virtual machine, an optimized JVM for embedded systems. The simulated devices in VisualSense are Java classes themselves that run on the standard JVM (Hotspot or equivalent), which also runs the simulator itself [32]. The same JVM is also used in Callas UDP. To improve maintenance, most of the code for the CVM implementation is shared between platforms. The platform specific code for the CVM in the SDK is limited to the components that interact with the network (communication) and with the operating system (hardware). The shared codebase is divided into two parts: the *interpreter* and the *bytecode manipulation*. The former consists of data structures representing the state of the virtual machine, and implements the reduction rules presented in Section 3. We rely entirely on Java's garbage collector for memory management. This choice is adequate for a proof-of-concept implementation. The interpreter is a switch statement on the next instruction being executed, where each case implements a different rule from the operational semantics. The bytecode manipulation part includes a parser that loads a bytecode program to create the runtime data structures needed to start the interpreter, and is also responsible for marshaling and un-marshaling Callas values for network communication.

4.1. Support for the SunSPOT platform

We have implemented a Callas virtual machine on top of the Squawk virtual machine [31] for SunSPOT devices [23] that supports the execution of Callas applications in this WSN platform. The basic SunSPOT device has a three-axis accelerometer, temperature and light sensors, eight multicolored LEDs, two push-button control switches, five digital I/O pins, six analog inputs, and four digital outputs. ZigBee is used for wireless communication. The virtual machine is modular so that only the networking and the access to the sensor board is distinct from the other instances. The code specific for implementing the CVM in SunSPOT devices is about 400 lines of Java code. The network layer is responsible for: marshaling and broadcasting messages; and receiving and unmarshaling messages. We expose the operating system/library calls that allow the access to the existing sensors and actuators in SunSPOT devices.

The architecture of the runtime system can be seen in Fig. 10. The full program includes three threads, one that runs the interpreter, one that receives messages from the network, and one that sends messages to the network. The communication model of the virtual machine is very akin to middleware systems except that calls are obviously asynchronous. The main thread interleaves the execution between: (a) the interpreter, that evaluates terms and triggers timed-calls, (b) placing function calls produced by the thread receiving data in the input queue of the interpreter; and (c) transferring all function calls in the interpreter's output queue to the thread transmitting data. The thread responsible for receiving

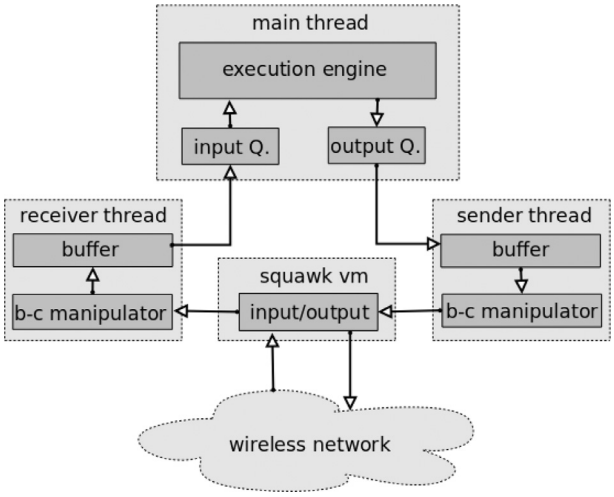

Fig. 10. The Callas runtime.

data uses the network infrastructure to accept bytecode messages, and unmarshals them into passivated function calls that are subsequently consumed by the main thread. Finally, the thread transmitting data to the network consumes passivated function calls, produced by the main thread, marshals them into bytecode messages, and broadcasts them. As we have said, low-level network communication is handled by the Squawk virtual machine that uses the ZigBee 802.15.4 wireless protocol intended for low-speed and low-power communication between devices.

4.2. Support for the VisualSense platform

One of these supports the execution of Callas applications in the WSN simulation tool VisualSense [24], developed as part of the U.C. Berkeley's project Ptolemy.

The SDK for simulating Callas applications using VisualSense consists of two applications (besides the Callas compiler): a generator of simulation models, and the runtime system that extends VisualSense to support Callas applications. Our compiler currently uses a deployment file, called the *network file*, to describe the code that runs on each category of devices, the signature of the existing remote function calls, and the signature of the available operating system interface. The network file is an extensible format and allows for more metadata to be included. The model generator uses a network file with simulation specific parameters to generate a MoML file, the XML-based format used by VisualSense, with a simulation model. In VisualSense, adding specific support for simulated devices running the CVM requires an implementation work similar to the one performed for SunSPOTs, namely, writing the network-related code, the operating system interface, and the intermediary code that executes the CVM in the simulator. The component that exposes the CVM as a simulation element amounts to 900 lines of code.

Fig. 11 shows an example of a Callas deployment file. The file defines network level and device level directives. The network directives include, for instance, the network interface type, which specifies the signature of the functions devices may contain, or the `CallasPowerLossChannel.lossProbability`, which redefines the `lossProbability` parameter of the `CallasPowerLossChannel`, itself an extension of VisualSense's `PowerLossChannel` (an abstraction for a type of wireless channel). Each `device` definition encloses a set of parameters that determines the common properties of a group of devices. The `code` parameter specifies the file with the Callas program that implements the behavior of this type of devices; the `size` parameter tells the generator how many devices run the specified `code`, and the `template` is used to get the device's MoML model. The `range` parameter determines the device's transmission range and the `position` specifies how they are distributed in the field. It can have three forms:

- explicit x_1, y_1 ... [x_n, y_n], explicitly defines the positions for all the devices;

```
# file: networkReadyForSimulation.calnet

interface = iface.caltype

template = network.moml
CallasPowerLossChannel.lossProbability = 0.0

device:
  code = node.callas
  size = 100
  range = 250
  position = random 0,0 to 1000,1000
  template = genericNode.moml
  Clock.period = 0.001

device:
  code = sink.callas
  size = 1
  range = 5000
  position = explicit 0,0
  template = genericNode.moml
  Clock.period = 0.001
```

Fig. 11. Callas deployment file.

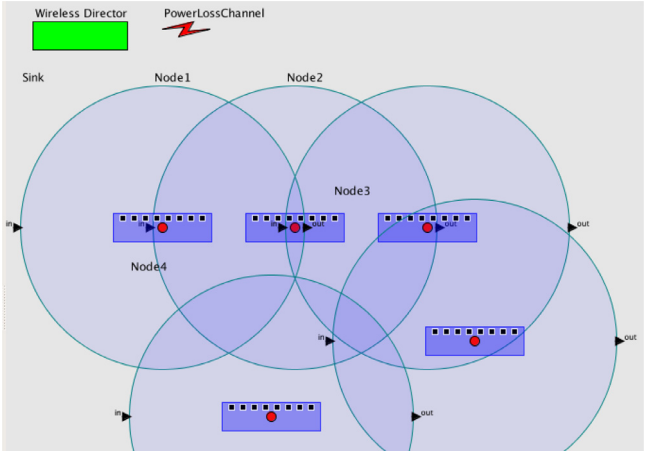

Fig. 12. Snap-shot of Callas application running on VisualSense. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

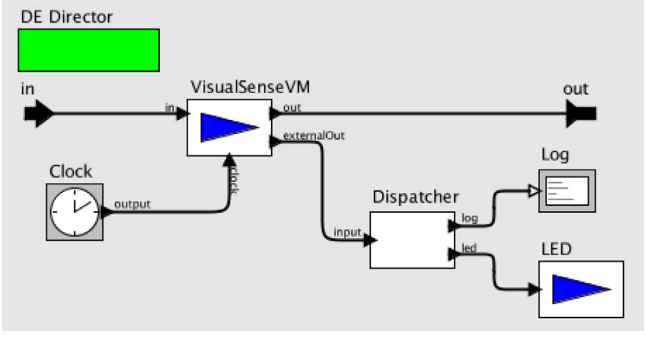

Fig. 13. Snap-shot of Callas application running on VisualSense.

- uniform x_1, y_1 to x_2, y_2, uniformly distributes devices within a bounding box defined by the arguments;
- random x_1, y_1 to x_2, y_2, like the former, but with a random distribution.

In Fig. 12 we present a network model consisting of a Wireless Director and a PowerLossChannel. Each device is represented in the GUI by a (blue) box with a (red) spot on the center marking its exact position, and a row of (black) squares, modeling a LED array. The wide (light-blue) circle around the device represents its transmission range. In this representation, it is possible to perceive that the lowermost device is isolated from the other devices: its transmission range is insufficient to reach any of them, and the others cannot reach it as well.

Each device is represented in VisualSense as a composite actor containing a set of (atomic) actors that define its behavior, as depicted in Fig. 13. Next we describe briefly each actor. A device has an input port `in`, and an output port `out`, bound to the channel simulating signal transmission in VisualSense, mimicking the device's antenna. The `VisualSenseVM` actor is a wrapper for the CMV, parameterized by: the Callas VM; the set of supported operating system calls; and the path to a Callas bytecode file. It converts the tokens received in its `in` port to the CVM message format and places them in the CVM's input queue. Conversely, the messages on the CVM output queue are converted to VisualSense tokens and sent to the actor's `out` port. This actor also receives input from a clock that sets the pace of the device. Each device has its own clock, allowing to simulate a network where devices run at different clock rates. Each clock tick period may be set in the deployment file and, for instance, setting the tick period to 0.01 s yields a 100 Hz hardware clock. Operating system calls may have

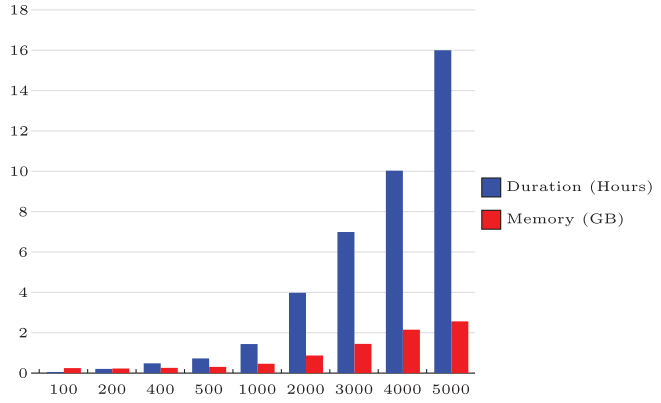

Fig. 14. Duration of simulation and memory usage (horizontal axis) given the number of sensors in the network (vertical axis).

side effects, i.e., they may influence other hardware components. For that, we included a `Dispatcher` that receives tokens from the VisualSenseVM `externalOut` port and forward these information to the actors that simulate each side effect from the native operations. We decided to include an actor, `LED`, that graphically represents a set of LEDs that can be turned on and off, simulating in this way a hardware side effect from the CVM. The `Log` actor writes messages to, for instance, a GUI window, a file, or a database. It is included only for debugging purposes.

We performed a sequence of simulations of the running example presented in Section 2 for 10 min (device time), in which we varied the amount of devices in the network, as depicted in Fig. 14. The results were obtained with VisualSense 7.01 on a Linux based PC with an Intel QuadCore 2.66 GHz CPU and 3.4GB of RAM. Our experiments show that the simulation duration grows polynomially while the memory footprint grows linearly. We believe that simulation duration is not a critical factor, as one would expect to wait for a few hours before having results for a 5000 device network. Moreover, an inspection of the simulated application reveals that the number of messages flowing on the network grows exponentially with the increase of the number of devices.

4.3. CallasUDP

The final Callas instance supports the execution of Callas applications over LANs, using processes to simulate devices and UDP/IP messaging with software filters to simulate wireless channels. This is intended as a fast prototyping and debugging tool for Callas applications that also has potential as a pedagogical tool.

5. An integrated development environment for Callas

From the beginning of the Callas project, we felt the need to implement applications within a programming environment that would make the development cycle shorter [33]. We envisioned deploying Callas applications over many distinct platforms, and not necessarily just physical sensor networks. This programming environment would then ideally integrate the Callas compiler and the Callas virtual machine seamlessly and provide for application deployment with minimal hassle. The IDE we had in mind had the following functional requisites: (a) *Editing and re-factoring.* The usual functionality such as syntax highlight, line and column number annotation, re-factoring, finding occurrences of identifiers; (b) *Project management.* Creating, configuring, managing, viewing/browsing multiple file projects; (c) *Building and Deploying.* Automatic and on-demand compilation of projects, viewing compilation errors, "one-click" deployment of executable files, full integration with the Callas compiler and with the Callas virtual

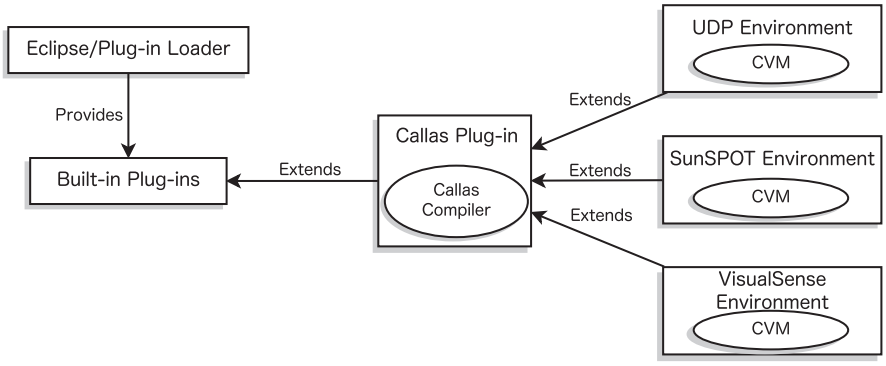

Fig. 15. The architecture of the Callas Eclipse Plugin (from [33]).

machine, support for multiple target platforms (c.f., Fig. 9); and (d) *Software Releases.* Support the release of the Callas programming language and targets as a single Eclipse plugin, with all the functionality, demos, and documentation embedded.

The Eclipse IDE [28] was used as the basis for the development as it provides a rich set of abstractions, templates, and tools that enable the fast prototyping of plugins. Moreover, the Callas compiler and virtual machine had been, up to that moment, fully developed within Eclipse, something that also weighted in the decision. Eclipse is structured around a core runtime system that implements all the basic functions of the IDE, and uses plugins to introduce new functionality. It also includes its own tool to aid in plugin development.

The structure of the plugin is straightforward (Fig. 15). It grows from the core Eclipse IDE by extending appropriate modules/plugins, e.g., Wizards, Properties, Editor, to name a few, to support the Callas programming language. Each of these plugins provides an extension point that allows its functionality to be modified or extended by user defined plugins. The Callas plugin registers itself as an Eclipse plugin by writing an appropriate entry in a XML configuration file. The file is read when Eclipse starts and context information (e.g., workspace location) is passed on to the Callas plugin Activator class. This information is used by all the plugin modules. These are activated, as required, at runtime, based on the type of project or file extension being used.

Wizards provide graphical interfaces that help users to create projects, files, or folders. When projects are created the plugin marks them as Callas projects, which allows the appropriate plugin to be called when these projects or files are manipulated. *Properties* is a component that is responsible for associating and managing attributes for each project. The attributes contain relevant information for the execution of the plugin, e.g., the identification of a Callas project's target platform. *Preferences* provides graphical tools to manipulate editor attributes, e.g., the color or size of the language reserved keywords or the periodicity of the project builder. The *Editor* is a fundamental part of the plugin, since programmers will use it exhaustively (Fig. 16). It takes care of syntax highlighting, automatic completion, block skeletons, and search for identifiers in a project. It also interacts with the builder by signaling changes to the source code and issuing build requests.

The main novelty is in the way the Callas compiler and runtime system are embedded in the Callas plugin and the modular support for multiple target platforms. The *Builder* extends the Eclipse builder plugin to allow the compilation of Callas applications and to provide instant compiler feedback to the programmer, the output being redirected to the "Problems View" window in Eclipse (Fig. 16, bottom). The compiler has been integrated into the Callas plugin through the class `CallasBuilder`. Fig. 17 shows a sample of the relevant code for the plugin. This class uses an abstraction for the Callas compiler and virtual machine, interface `CallasAPI`,

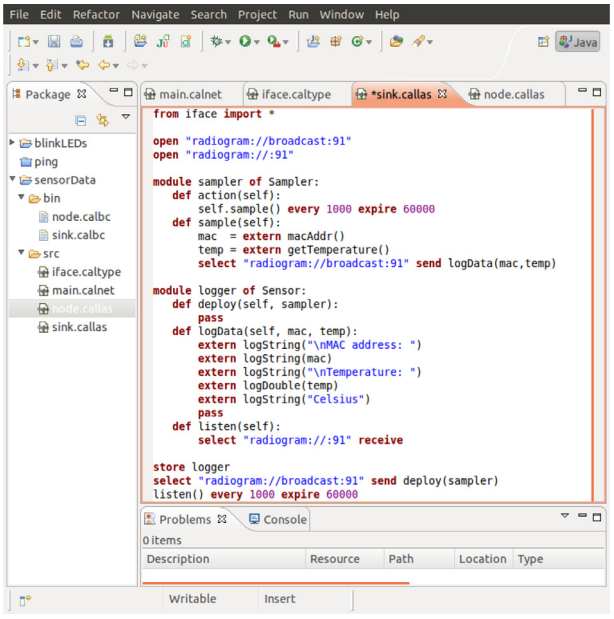

Fig. 16. Snapshot of the Callas Editor (with new version of the syntax, from [33]).

to get access to some modules of the Callas compiler, namely the parser and the type-checker. The `build` method accesses some properties of the current Callas project, namely its description file and the corresponding path, and runs the parser and type-checker on the project's source files.

The build method can be called in two different ways. First, it may be used on demand by the programmer, by simply selecting the option from a menu or clicking on the appropriate icon. The *Builder* can also be configured to run periodically in the background checking for changes in the source files of the current project. This is implemented as a lightweight Java `TimerTask` class with a user configurable period. The task periodically checks if the project is "dirty" and, if so, it invokes the build method on the modified sources.

The *Launcher* extends the functionality of the Eclipse "launch" plugin and deploys the Callas application resulting from building a project on a configurable target platform, e.g., a network of SunSPOT devices. When the launcher is activated for a Callas project, the Eclipse "launch" plugin passes the control to the Callas plugin launcher. The latter also provides an extension point, to allow for multiple platforms to be supported. Adding support for a new platform works exactly like extending Eclipse plugins. The user provided, platform specific, launchers register themselves in the Callas plugin launcher as new modules and are automatically loaded when a Callas project exhibits the appropriate properties.

```
class CallasBuilder extends IncrementalProjectBuilder {
 ...
 void build(int kind, Map args, IProgressMon mon) {
   ...
   CallasAPI api = new CallasAPI();
   String calnet = getProject().getProperty("CALNET");
   String path = getProject().getProperty("PATH");
   InputStream code = getInputStream(calnet, path);
   NetworkFile netFile = api.parser().parse(calnet, code);
   api.typechecker().typecheck(netFile);
   ...
 }
 ...
}
```

Fig. 17. Integration of the Callas compiler in the Callas plugin.

```
class SunSPOTEnv extends ... implements ICallasRunTimeEnv {
 ...
 void launch(ILaunchConf conf, String mode, ILaunch launch,
             IProgressMon mon) {
  ...
  String bytecode = conf.getAttribute("BYTECODE");
  String target   = conf.getAttribute("TARGET");
  String antCmd   = mkAntCmd(target, bytecode);
  Process child   = DebugPlugin.exec(antCmd, mon);
  DebugPlugin.newProcess(launch, child, ...);
  ...
 }
 ...
}
```

Fig. 18. Part of the SunSPOT specific launcher implementation.

Fig. 18 shows a sample of the code for a platform specific launcher for the SunSPOT devices. When the programmer selects the "Run" option, Eclipse detects that it is a built Callas application and loads the Callas plugin launcher. In turn, this launcher asks the programmer to specify the target from a list of registered platforms. With this information at hand, it then redirects the operations to the platform-specific launcher that implements the interface `ICallasRuntimeEnv` featuring the main method `launch`. When the method is called we get the properties of the project with the bytecode and the target and execute appropriate commands to deploy the application. The `DebugPlugin` class is used to run the aforementioned commands and to redirect their output to the Eclipse console.

6. Related work

There are many proposals for programming languages for WSN in the literature. Such languages provide different levels of abstraction from the hardware as well as different views of the network and of the computational agents. A compromise between abstraction level and resource availability is always implicit in the proposals with, in general, higher level programming languages requiring more hardware and energy resources from the nodes in an WSN. At the lower range of resource availability one finds systems like Pushpin [8]. This is an extremely lightweight system as there is not even an abstraction layer for programming. Pieces of native code called *pfrags* are transferred and executed in the nodes (called *pushpins*). A tiny operating system provides a shared memory address space for communication between pfrags and a few basic system calls. Mottle [10] provides powerful primitive data structures and first class functions. It is implemented on top of the MatÃ©virtual machine, developed for TinyOS based WSN. Mottle programs, called *capsules*, are compiled into MatÃ©assembly code and, as such, may be injected in the network at any time to perform specific tasks. Capsules have the capability of moving between sensor nodes, a form of code mobility. nesC [11] is a programming language developed on top of the TinyOS [7] operating system. Programs are nested collections of components, some of which may be provided by TinyOS itself. An application is built from several components that are linked into an executable that is run by the underlying TinyOS engine as a non-preemptive tasks. Besides running tasks, TinyOS also captures events and directs them to the appropriate handling code.

Some languages provide programmers with very high-level views of the network by hiding all networking and communication details. The programmer implements a distributed application that usually is not targeted at a specific sensor network architecture or configuration. A specialized compiler takes this high-level view of the application and produces the node specific behavior for each sensor as required for the deployment of the application, without the intervention of the programmer. This approach is called *macro-programming*. A good example of such a language is Regiment [34].

It uses network *regions* and data *streams* as the basic programming abstractions. The run-time for the language is based on a distributed version of a token machine (DTM). Nodes perform sensing and computation in response to tokens received from the network or to tokens generated internally. Other languages also adhere to the region abstraction as described. Abstract Regions [15], implemented on top of TinyOS, implements a model of regions that supports the discovery of neighborhood nodes and the sharing of data among nodes and regions. No primitives for data aggregation nor for fine-tuning the trade-off between accuracy and resource consumption are given.

Other languages take a data-centric approach, viewing sensor networks as data repositories upon which standard database processing primitives may be used. One example is TinyDB [13], a macro-programming system that allows a programmer to reason about a sensor network as a database. Accordingly, the user may write declarative style queries to request a particular *view* of the data being generated by the network. This is done at a very high-level of abstraction without the user having any notion of the underlying network. A sophisticated compiler decomposes the user queries into low-level, sensor specific operations based on primitives like sampling, application of filters to data, data aggregation, and data broadcast. A similar approach is taken in Cougar [14], which allows users to specify high-level queries for data views to be extracted from a sensor network. The system then analyses the queries and decomposes them into a sequence of network operations optimized to minimize resource consumption. Another approach is used in Sense2P [35], a logic macroprogramming system for abstracting and programming WSNs as globally deductive databases, with simplicity and performance advantages over using SQL-type queries over data generated from a WSN. Still other projects focus on the interface between the Internet and the sensor networks viewed as Web resources. In this line, IrisNet [18] is a Java based system that allows programmers to specify distributed database services through XML documents and provides a set of high-level APIs to compose data queries to sensor networks and to process the resulting data.

Finally, other languages introduce mobile agents in WSN. An example of this approach is Agilla [17], in which programs are autonomous agents that move between the nodes in a sensor network. Each agent runs on top of a virtual machine. The communication model is based on a distributed Linda tuple-space. Network re-programming is allowed by injecting new agents into the network and by killing existing ones. One of the goals of Agilla is to transform more resourceful sensor networks into general purpose computing platforms. A similar approach is taken by SensorWare [16], a system that, given its size, aims at more resourceful sensor nodes and networks. Programs appear as mobile scripts and a sensor node is seen as a dynamically changing entity where new scripts may be installed on the fly. The scripts may be injected in the network at any time. The language is TCL-based with primitives for timer services, acquisition and sensing data, mobility of scripts, and for a location discovery protocol.

Other approaches focus on providing programming languages for WSN that are friendly to non-specialists [36–38]. Although the technology is widely available and interesting to many fields, application developers do not usually have the skills or experience to dwell in low-level programming and hardware configurations. Absynth [36] is a project that explores this line of research by providing archetypes for common network configurations and applications, including a simple programming language, e.g., Wasp [39], and application templates. A similar approach is taken by Sonar [37] in providing developers with a seamless environment to deploy WSN, a simple domain-specific programming language and virtual machine, and a tiny operating system, allowing for the dynamic reprogramming of WSN.

7. Conclusion and future work

In this paper we address the problem of providing WSN with programming languages that eliminate some types of runtime errors, aiming to simplify the debugging and the deployment of applications. Our main argument is that this can be achieved by carefully designing programming languages and their runtime systems, to be *safe-by-design*. Type-safe languages, for instance, allow programmers to develop applications that are guaranteed to be free of runtime errors such as misuse of interfaces. Well-typed applications never produce such runtime errors. From the point of view of the language runtime system, the existence of a formal specification, e.g., in the form of an abstract machine, allows the verification of its soundness, i.e., that it preserves the operational semantics of the language. In this case, applications never produce runtime errors due to a faulty design of the runtime system. Another issue, which is the subject of ongoing work, refers to the language compiler and whether it generates code that preserves the semantics of the programming language. This would guarantee that applications would not incur in runtime errors due to errors in the generation of code by the compiler. These design principles eliminate the major sources of subtle semantic errors from WSN applications and provide a typed programming discipline that allows the premature detection of would-be runtime errors.

To provide a proof-of-concept for the aforementioned design principles, we implemented a new type-safe programming language, Callas, based on a formal model for computations in sensor networks. The runtime for the language was specified in the form of an abstract machine and it soundness relative to the operational semantics of the language was asserted. We describe the language compiler and three target platforms that are currently supported by the framework: networks of SunSPOT devices, the simulation environment VisualSense, and a testbed implementation over local area networks using UDP datagrams to simulate wireless channels. Finally, to provide a complete programming environment, we developed a plugin for the Eclipse IDE that allows Callas programmers to implement applications and to deploy them over the supported target platforms in a seamless way. In fact, the plugin environment is being used to provide the Callas software releases, with embedded language compiler, runtime systems, and platform specific software. The prototype here described and documentation for the project can be downloaded from the Callas Project Homepage [40].

In terms of future work, we are interested in proving that the Callas compiler produces correct byte-code, i.e., preserving the operational semantics of the language. We are interested in developing higher level idioms for programming WSN (e.g., a macroprogramming language) fully encoded in Callas therefore inheriting its safety properties.

Acknowledgments

This work was sponsored by project MACAW (Fundação para a Ciência e Tecnologia contract PTDC/EIA-EIA/115730/2009) and by project "NORTE-07-0124-FEDER-000058" (SENSING) financed by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia.

References

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, A survey on sensor networks, IEEE Commun. Mag. 40 (8) (2002) 102–114.
[2] L. Lopes, F. Martins, J. Barros, Middleware for Network Eccentric and Mobile Applications, Springer-Verlag, pp. 25–41.

[3] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms, ACM/Kluwer Mobile Netw. Appl. (MONET) Spec. Issue Wireless Sens. Netw. 10 (4) (2005) 563–579.
[4] A. Dunkels, B. Grönvall, T. Voigt, Contiki—a lightweight and flexible operating system for tiny networked sensors, in: First IEEE Workshop on Embedded Networked Sensors (EmNets'04), Tampa, Florida, USA, 2004, pp. 455–462.
[5] A. Eswaran, A. Rowe, R. Rajkumar, Nano-RK: an energy-aware resource-centric operating system for sensor networks, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS'05), 2005, pp. 256–265.
[6] C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (MobiSys'05), ACM Press, New York, NY, USA, 2005, pp. 163–176.
[7] The TinyOS Documentation Project, 2004–present, http://www.tinyos.net.
[8] J. Lifton, D. Seetharam, M. Broxton, 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, pp. 139–151.
[9] P. Levis, D. Culler, Maté: a tiny virtual machine for sensor networks, in: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), ACM Press, 2002, pp. 85–95.
[10] P. Levis, D. Gay, D. Culler, Bridging the Gap: Programming Sensor Networks with Application Specific Virtual Machines, Technical Report UCB//CSD-04-1343, University of California at Berkeley, 2004.
[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, The nesC language: a holistic approach to network embedded systems, in: ACM Conference on Programming Language Design and Implementation (PLDI'03), 2003, pp. 1–11.
[12] R. Newton, Arvind, M. Welsh, Building up to macroprogramming: an intermediate language for sensor networks, in: Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'05), 2005, pp. 37–44.
[13] S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, TinyDB: an acquisitional query processing system for sensor networks, ACM Trans.Database Syst. 30 (2005) 122–173.
[14] W.F. Fung, D. Sun, J. Gehrke, COUGAR: the network is the database, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD'02), ACM Press, 2002, p. 621.
[15] M. Welsh, G. Mainland, Programming sensor networks using abstract regions, in: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04), 2004, p. 3.
[16] A. Boulis, C. Han, M.B. Srivastava, Design and implementation of a framework for efficient and programmable sensor networks, in: Proceedings of the First International Conference on Mobile Systems, Applications and Services (MobiSys'03), ACM Press, 2003, pp. 187–200.
[17] C.-L. Fok, G.-C. Roman, C. Lu, Rapid development and flexible deployment of adaptive wireless sensor network applications, in: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05), IEEE Press, 2005, pp. 653–662.
[18] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, S. Seshan, IrisNet: an architecture for a world-wide sensor web, IEEE Pervasive Comput. 2 (4) (2003).
[19] L. Lopes, F. Martins, M.S. Silva, J. Barros, A process calculus approach to sensor network programming, in: Proceedings of the International Conference on Sensor Technologies and Applications (SENSORCOMM'07), IEEE Computer Society, 2007, pp. 451–456.
[20] K. Honda, M. Tokoro, An object calculus for asynchronous communication, in: Proceedings of the European Conference on Object-oriented Programming (ECOOP'91), in: LNCS, vol. 512, Springer-Verlag, 1991, pp. 133–147.
[21] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes (parts I and II), Inf. Comput. 100 (1992) 1–77.
[22] F. Martins, L. Lopes, J. Barros, Towards safe programming of wireless sensor networks, Electron. Proc. Theor. Comput. Sci. 17 (2010) 49–62.
[23] Project Sun SPOT, 2004–present, http://www.sunspotworld.com.
[24] P. Baldwin, S. Kohli, E.A. Lee, X. Liu, Y. Zhao, Modelling of sensor nets in Ptolemy II, in: Proceedings of the Third International Symposium on Information Processing in Sensor Networks (IPSN'04), ACM Press, 2004, pp. 359–368.
[25] K.V.S. Prasad, A calculus of broadcasting systems, in: International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91), in: LNCS, vol. 493, Springer-Verlag, 1991, pp. 338–358.
[26] K. Ostrovský, K.V.S. Prasad, W. Taha, Towards a primitive higher order calculus of broadcasting systems, in: International Conference on Principles and Practice of Declarative Programming (PPDP'02), ACM Press, 2002, pp. 2–13.
[27] N. Mezzetti, D. Sangiorgi, Towards a calculus for wireless systems, in: Twenty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS'06), in: ENTCS, 158, Elsevier Science, 2006, pp. 331–354.
[28] Eclipse IDE, 2001–present, http://www.eclipse.org.
[29] R. Harper, B. Pierce, A record calculus based on symmetric concatenation, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91), The ACM Press, 1991, pp. 131–142.
[30] T. Cogumbreiro, P. Gomes, F. Martins, L. Lopes, Safe-by-Design Programming Languages for Wireless Sensor Networks, technical report DCC-2011-09, Department of Computer Science, Faculty of Sciences, University of Porto, 2011.Available at http://www.dcc.fc.up.pt/dcc/Pubs/TReports/.
[31] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, 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), 2006, pp. 78–88.
[32] D. Vieira, F. Martins, Automatic generation of WSN simulations: from Callas applications to visualsense models, in: Proceedings of the 2010 Fourth International Conference on Sensor Technologies and Applications (SENSORCOMM'10), IEEE Computer Society, 2010, pp. 336–341.
[33] J. Torres, An Integrated Development Environment for the Callas Programming Language, Department of Computer Science, Faculty of Sciences, University of Porto, 2011 Master's thesis.
[34] R. Newton, M. Welsh, Region streams: functional macroprogramming for sensor networks, in: First International Workshop on Data Management for Sensor Networks (DMSN'04), Toronto, Canada, 2004, pp. 78–87.
[35] S. Choochaisri, N. Pornprasitsakul, C. Intanagonwiwat, Logic macroprogramming for wireless sensor networks, Int. J. Distrib.Sens. Netw. 2012 (2012) (pages 12) http://www.hindawi.com/journals/ijdsn/2012/171738.
[36] The Absynth Project, 2007–present, http://absynth-project.org/.
[37] G. Ferro, R. Silva, L. Lopes, Towards out-of-the-box programming for wireless sensor networks, in: Proceedings of the 18th IEEE International Conference on Computational Science and Engineering (CSE2015), Porto, Portugal, IEEE Computer Society, 2015.
[38] A. Elsts, J. Judvaitis, L. Selavo, SEAL: a domain-specific language for novice wireless sensor network programmers, in: Proceedings of the 39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'13), Santander, Spain, IEEE Computer Society, 2013, pp. 220–227.
[39] L.S. Bai, R.P. Dick, P.A. Dinda, Archetype-based design: sensor network programming for application experts, not just programming experts, in: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks, in: IPSN'09, IEEE Computer Society, 2009, pp. 85–96.
[40] The Callas Project, 2008–present, http://www.dcc.fc.up.pt/callas.

Luís Lopes got his Ph.D. on Computer Science from the University of Porto, in 1999. His research interests include domain specific programming languages, virtual machines, distributed systems, embedded systems and, in particular, wireless sensor networks. He is an associate professor at the Department of Computer Science at the Faculty of Science, University of Porto.

Francisco Martins is an assistant professor at the Department of Informatics, Faculty of Sciences, University of Lisbon. Until September 2006 he was assistant professor at the Department of Mathematics, University of Azores where he began teaching (as teaching assistant) in October 1997. Previously he was an I. T. manager at Banco Comercial dos Açores since 1990. He received his Ph.D. in Computer Science at University of Lisbon (Faculty of Sciences) in 2006, his M.Sc. (by research) in Computer Science at University of Azores in 2000, and his B.Sc. in Mathematics and Informatics at the University of Azores in 1995.