A Semantically Robust Framework for Programming Wireless Sensor Networks

Luís Lopes
CRACS & INESC-Porto/DCC-FCUP, Portugal
lblopes@dcc.fc.up.pt

Francisco Martins
LASIGE/DI-FCUL, Portugal
fmartins@di.fc.ul.pt

Technical Report Series: DCC-2010-01

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

A Semantically Robust Framework for Programming Wireless Sensor Networks

Luís Lopes
CRACS & INESC-Porto/DCC-FCUP, Portugal
lblopes@dcc.fc.up.pt

Francisco Martins
LASIGE/DI-FCUL, Portugal
fmartins@di.fc.ul.pt

Abstract

Programming and debugging wireless sensor networks is a hard task, not only due to the fragility and volatile nature of the hardware, but also, on a very fundamental level, due to deficiencies in the current programming tools. In this paper we propose a distinct approach to the design and implementation of programming languages for wireless sensor networks. We argue that such languages should be type-safe as this allows an appropriate compiler to prematurely (statically) detect programs that otherwise would produce run-time errors. Moreover, we argue that the specifications for the language run-times should be proved to preserve the language semantics, thus removing a semantic gap that can be observed in current systems. We present a type-safe core programming language for sensor networks called Callas and provide a specification for a virtual machine for it. We prove that the virtual machine preserves the programming language semantics. This establishes Callas as a robust semantic framework for designing and implementing higher level programming languages for wireless sensor networks.

keywords: Wireless Sensor Networks, Type-Safety, Soundness, Virtual Machine.

I. INTRODUCTION

Wireless sensor networks are collections of potentially large number of physical devices capable of measuring environmental variables, quantify those measurements, and transmit them to one or more network sinks where the data is gathered and eventually processed. Data transmission is accomplished over radio links and routing based on ad-hoc networking protocols [1]. Sensor networks have several particularities that set them apart from other ad-hoc networks such as Mobile Ah-doc NETworks (MANETS) that result in distinct technologies for wireless communication, networking protocols, and programming tools. The network devices have severe hardware limitations. They typically feature a microprocessor with limited processing power, a very limited amount of application memory, and a battery for power. The fact that they are often deployed in remote locations puts stringent constraints on the duty-cycle of the battery and in fact, all aspects of the development of wireless sensor networks are *energy centric*. This implies that applications for wireless sensor networks must be *lightweight*, have a *small memory footprint*, and be *power conservative*. Another consequence of their physical remoteness is the fact that access to the devices (*e.g.* for maintenance, debugging, or re-programming) is difficult or even impossible. *This argues for programming languages that provide both network re-programming, in order to apply patches for bugs, and statically verified executables, in order to prematurely catch would-be run-time errors*. From a programming point of view, sensor networks are often designed for specific applications or application domains making software re-usability and portability an issue. *This argues for the development of run-time systems capable of abstracting away from the device's hardware and operating system, e.g. based on virtual machines, whose specifications may be subject to verification*.

The *semantic robustness* we argue for in this paper has two components. First, the programming language must be type-safe. Type safety ensures that well-typed programs do not give rise to a certain class of run-time errors. A well designed language and type system can provide the programmer with a compiler capable of checking code in this fashion. This language feature allows many possible run-time errors to be caught statically at compile time, before the application is deployed on the network. The other way of providing robustness to applications is to ensure that the underlying run-time system preserves the semantics of the programming language. Thus, to

minimize the possibility of execution errors, the run-time system must be implemented based on a specification that, it can be proven, matches the semantics of the programing language. While this still leaves some margin for errors introduced by the programmer of the run-time, these may be eliminated through an exhaustive evaluation and test of the software.

There are currently many programming languages proposed for wireless sensor networks [8]. These languages range from the rather low-level Pushpin [6], to intermediate, as is the case of the ubiquitous nesC [4], and to high-level idioms based on abstractions such as streams in Regiment [15], databases in TinyDB [10], and agents in SensorWare [3]. Despite the diversity of approaches, to our best knowledge, none of the above programming languages tackles the *semantic robustness* issue. For example, Regiment [15], a strongly-typed functional *macro-programming* language, is the closest to achieve this goal by providing a type-safe compiler. However, Regiment is then compiled into a low-level *token machine language* that is not type-safe. This intermediate language is itself compiled into a nesC implementation of the run-time based on the *distributed token machine* model, for which no correctness properties are available.

In [9] we introduced Callas, a calculus based on the formalism of process calculi [5, 14] to provide a very basic, assembly-like language for programming wireless sensor networks. Elsewhere we have established that this model, coupled with an appropriate type-system, is type-safe [11]. The type-system filters out programs with wrong interaction protocols (e.g. sending a message with the wrong number or type of arguments). As a side note, the assembly-like nature of the language is not restrictive in anyway. Higher level programming can be done on top of the this core language by providing high-level abstractions encoded as sequences of core processes. These encodings must of course preserve the semantics of the core language. A compiler just pre-processes the higher level program into a core language before producing an executable file. A few examples of these encodings are presented in the next section.

In this paper, we go one step further and provide a full specification for a virtual machine for the programming language and prove that it preserves the language semantics. This establishes Callas as a semantically robust framework on which to develop applications for sensor networks. We have implemented a preliminary language compiler and prototype of the virtual machine for the SunSPOT platform that will allow us to experiment with the language and perform large scale simulations (with slight reconfiguration) using existing tools (*e.g.* [2]).

The remainder of the paper is structured as follows. Chapter II presents the Callas model: its operational semantics and basic type-safety results. Chapter III presents a specification for the Callas virtual machine, followed in Chapter IV by the proof that the specification is correct relative to the operational semantics of the language. Chapter V ends the paper with some conclusions and perspectives for future work.

II. THE PROGRAMMING MODEL

Callas [9, 11] is a language for programming sensor networks that provides primitives for sensor computation, communication, code mobility, and code updates.

The language abstracts away from the physical, link, and network layers of the protocol stack. The focus is entirely on the application layer and we assume that low-level networking and transmission details are handled by underlying software and hardware. The syntax of the language is given in Fig. 1.

A. Syntax

Programs P, the computational elements of the sensor devices, are expressions that may call a function l in a module v with arguments $\vec{v}$ ($v.l(\vec{v})$); alternatively, it may call a function external to the application, *e.g.* a function from the operating system (**extern** $l(\vec{v})$). In a timed-call (**timer** $l(\vec{v})$ **every** v_1 **expire** v_2), the call to function l ($l(\vec{v})$) is executed periodically (every v_1) during a time interval (v_2), after which the timer expires. Communication is accomplished by sending (**send** $l(\vec{v})$) and receiving messages (**receive**). Sending a message means to asynchronously invoke a remote function in the neighborhood sensors; incoming calls are placed in a queue and processed when the sensor explicitly accepts the message (**receive**). The **let** construct permits the definition of local variables and the processing of intermediate values. The latter is also useful to derive a basic sequential composition construct (in fact, $P; P' \stackrel{\text{def}}{=}$ **let** $x = P$ **in** P' with $x \notin \text{fv}(P')$). We make frequent use of this construct to impose a more imperative style of programming. Code may be updated by adding or replacing functions in modules (**update** M **with** M'). Expressions **load** and **store** are used to manipulate the sensor's

P	$::=$	*Programs*	e	$::=$	*Expressions*
	e	expression		v	value
	$\mid v.l(\vec{v})$	function call		$\mid e+e \mid e-e \mid e*e \mid e/e$	arith. exp.
	$\mid$ **extern** $l(\vec{v})$	external call		$\mid$ **not** $e \mid e$ **and** e	bool. exp.
	$\mid$ **timer** $l(\vec{v})$ **every** v **expire** v	timed call		$\mid (e) \mid e == e \mid e < e$	rel. exp.
	$\mid$ **send** $l(\vec{v})$	comm.	v	$::=$	*Values*
	$\mid$ **receive**	comm.		i	integer
	$\mid$ **let** $x = P$ **in** P	sequence		$\mid f$	float
	$\mid$ **update** v **with** v	code update		$\mid b$	boolean
	$\mid$ **load**	load		$\mid x$	variable
	$\mid$ **store** v	store		$\mid M$	module
	$\mid$ **if** b **then** P **else** P	conditional	M	$::= \{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$	*modules*

Fig. 1. The syntax of the Callas language.

S	$::=$	*Sensors*	R	$::= P_1 :: \cdots :: P_n$	run-queue
	0	empty net	T	$::= \{(l_i(\vec{v}_i), v_i, v_i, v_i)\}_{i \in I}$	timed calls
	$\mid S \mid S$	composition	m	$::= \langle l(\vec{v}) \rangle$	messages
	$\mid [P, R \triangleright M, T]^{I,O}_{p,t}$	sensor	I, O	$::= m_1 :: \cdots :: m_n$	I/O queues

Fig. 2. The syntax of Callas run-time environment.

internal memory. The **load** expression returns the sensor's memory contents as a module; in contrast, **store** v updates sensor's memory with module v. Finally, expression **if** b **then** P **else** P implements the usual conditional execution.

Expressions e allow for the combination and comparison of values in a standard manner.

Sensor devices exchange values v that include primitive data-types supported by the devices, notably integers i, floatings f, booleans b, and code modules M. Modules consist of sets of named functions, represented by $l = (\vec{x})P$, where l is the name of the function, $\vec{x}$ are the parameters, and P the code for the function.

Notice that the syntax allows for meaningless programs to be written, for instance calling a function on an integer value $1.l(\vec{v})$, however a type system (not shown) rules out such programs (*vide* [11]).

The run-time environment for Callas is presented in Fig. 2. Sensor networks S are concurrent compositions of sensors devices, represented as $[P, R \triangleright M, T]^{I,O}_{p,t}$, and of the empty network, denoted by **0**. Each device is composed by a running program P, a queue of pending programs 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), the current position p, and the current time t. Messages are passivated function calls denoted as $\langle l(\vec{v}) \rangle$.

B. Semantics

The operational semantics is defined with the help of a structural congruence relation (Fig. 3) as is usually with process calculi [13]. The only non-standard rule is $[P, R \triangleright M, T]^{I,O}_{p,t} \equiv [P, R \triangleright M, T]^{I,O}_{p,t} \{\mathbf{0}\}$ that provides the sensor with a conceptual *membrane* to prevent neighborhood sensors from receiving duplicate copies of a message during a broadcast [12].

The reduction relation is inductively defined by the rules in Fig. 4 and 5. Since programs evaluate to expressions, we allow for reduction within the **let** construct. To control the evaluation order we restrict reduction to always occur inside some **let** expression. Alternatively, we could present the semantics using reduction contexts, as we did in [11], but the current approach is more closely related to our byte code representation. The reduction steps are controlled by an internal clock t. The time for the next activation of every programmed timed call is check against

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \qquad S \mid \mathbf{0} \equiv S, \qquad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3$$
$$[P, R \triangleright M, T]^{I,O}_{p,t} \equiv [P, R \triangleright M, T]^{I,O}_{p,t}\{\mathbf{0}\} \qquad \text{(S-INIT-SEND)}$$

Fig. 3. Structural congruence for sensors.

the current clock time using the predicate $noEvent$. Reduction is driven by the running program P, executing the associated action and advancing 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'. Rules R-INTERRUPT and R-EXPIRE need to trigger all the calls and discard all the timers within the same time unit and hence do not advance the clock.

Rule R-EXTERN makes a synchronous call to an external function l and immediately receives a value v. The rules R-LOAD and R-STORE are used to access and to rewrite, respectively, the installed module in the sensor device. Rule R-SEND (R-RECEIVEM) handles the interaction with the network by putting (getting) messages in (from) the outgoing (incoming) queue. The **receive** operation is not blocking (Rule R-RECEIVEE) and the program progresses even when there is no incoming messages. Reduction occurs inside a **let** construct (Rule R-LETP), allowing a program P_1 to evolve into P_2. Eventually program P_1 evaluates to an expression e (Rule R-LETV), and the resulting value v replaces the free occurrences of variable x in P. When a programs evaluates to a value, the latter is discarded if there is a pending program in the run-queue, and that program becomes active (Rule R-NEXT). Otherwise, the sensor stalls until a program appears in the run-queue (Rule R-IDLE). Rule R-UPDATE handles module updates. It creates a new module that consists of the functions in M' plus the functions in M''. Duplicate functions are deleted in M' and replaced by the implementations in M''. Rule R-CALL handles calls to functions in modules. It selects the code for the function, replaces the parameters by the arguments, passing the current module M' as the first argument in variable *self*, and runs the resulting program.

The rule R-TIMER programs a timer for a call to a function installed in the sensor device (whose code is in M). When the $noEvent$ evaluates to false, the rule R-INTERRUPT comes into action. It places a timed function call $l(\vec{v})$ in the run-queue. The execution of the call is delegated to rule R-CALL. Finally, when the expiration time is reached the timer is discarded (Rule R-EXPIRE). The rules for the **if** instruction (omitted) are standard.

The reduction semantics for networks is orthogonal to that for in-sensor processing. Rules R-NETWORK and R-CONGR are straightforward. The former allows reduction to happen concurrently in sensor networks, while the latter brings the congruence relation into reduction. Communication occurs by broadcasting messages over a wireless channel to sensors in the neighborhood of the broadcasting sensor. Rule R-BROADCAST handles the distribution of outgoing messages by delivering such messages in the incoming queues of receiving devices. The semantics is parametric on predicates *inRange* and *networkRoute* that we leave unspecified (see the examples below). The *inRange* predicate checks whether a given device is within communication range of the current broadcasting device, using some criteria, *e.g.* geographic proximity, signal strength. The *networkRoute* predicate implements the network routing protocol used to propagate messages, *e.g.* mesh mode in the SunSPOT framework [16]. A transmission starts with the application of structural congruence rule S-INIT-SEND (Fig. 3), continues with multiple applications of R-BROADCAST, and terminates with an application of R-RELEASE.

Static semantics. In [11] we have introduced a type system for the Callas programming language and proved language type safety, meaning that well-typed programs do not produce a class of run-time errors, namely that (a) function calls are issued on a module that contains the function and that such calls match function's signatures; (b) a module update preserves the signatures of the functions it contains.

The following examples illustrate the programming paradigm. We use the following derived construct to install code in these examples:

$$\begin{aligned}\textbf{install}\ M \stackrel{\text{def}}{=}\ & \textbf{let}\ x = \textbf{load}\ \textbf{in} \\ & \textbf{let}\ y = \textbf{update}\ x\ \textbf{with}\ M\ \textbf{in} \\ & \textbf{store}\ y\end{aligned}$$

Streaming data I. The program that runs on the *sink* starts by installing a module with a receiver function and a gather function. The former just listens for messages from the network on the incoming queue. The latter

$$\frac{\text{noEvent}(T,t)}{[\langle \textbf{extern } l(\vec{v})\rangle, R \triangleright M, T]^{I,O}_{p,t} \to [\langle v\rangle, R \triangleright M, T]^{I,O}_{p,t'}} \qquad \frac{\text{noEvent}(T,t)}{[\langle \textbf{load}\rangle, R \triangleright M, T]^{I,O}_{p,t} \to [\langle M\rangle, R \triangleright M, T]^{I,O}_{p,t'}} \quad \text{(R-EXTERN,R-LOAD)}$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textbf{store } v\rangle, R \triangleright M, T]^{I,O}_{p,t} \to [\langle \{\}\rangle, R \triangleright v, T]^{I,O}_{p,t'}} \qquad \frac{\text{noEvent}(T,t)}{\begin{array}{l}[\langle \textbf{receive}\rangle, R \triangleright M, T]^{\langle l(\vec{v})\rangle :: I,O}_{p,t} \to \\ \qquad [\langle \{\}\rangle, R :: \textbf{let } x = \textbf{load in } x.l(\vec{v}) \triangleright M, T]^{I,O}_{p,t'}\end{array}}$$

(R-STORE,R-RECEIVEM)

$$\frac{\text{noEvent}(T,t)}{[\langle \textbf{receive}\rangle, R \triangleright M, T]^{\varepsilon,O}_{p,t} \to [\langle \{\}\rangle, R \triangleright M, T]^{\varepsilon,O}_{p,t'}} \qquad \frac{[P_1, R \triangleright M, T]^{I,O}_{p,t} \to [P_1', R' \triangleright M', T']^{I',O'}_{p,t'} \quad \text{noEvent}(T,t)}{[\langle P_1\rangle, R \triangleright M, T]^{I,O}_{p,t} \to [\langle P_1'\rangle, R' \triangleright M', T']^{I',O'}_{p,t'}}$$

(R-RECEIVEE,R-LETP)

$$\frac{\text{noEvent}(T,t) \qquad \text{eval}(e) = v \qquad x \neq y}{\begin{array}{l}[\textbf{let } x = e \textbf{ in } P, R \triangleright M, T]^{I,O}_{p,t} \to \\ \qquad [\textbf{let } y = P[v/x] \textbf{ in } y, R \triangleright M, T]^{I,O}_{p,t'}\end{array}} \qquad \frac{\text{noEvent}(T,t)}{[\textbf{let } x = v \textbf{ in } x, P :: R \triangleright M, T]^{I,O}_{p,t} \to [P, R \triangleright M, T]^{I,O}_{p,t'}}$$

(R-LETV,R-NEXT)

$$\frac{\text{noEvent}(T,t)}{[\textbf{let } x = v \textbf{ in } x, \varepsilon \triangleright M, T]^{I,O}_{p,t} \to [\textbf{let } x = v \textbf{ in } x, \varepsilon \triangleright M, T]^{I,O}_{p,t'}} \qquad \frac{\text{noEvent}(T,t)}{\begin{array}{l}[\langle \textbf{update } M' \textbf{ with } M''\rangle, R \triangleright M, T]^{I,O}_{p,t} \to \\ \qquad [\langle M' + M''\rangle, R \triangleright M, T]^{I,O}_{p,t'}\end{array}}$$

(R-IDLE,R-UPDATE)

$$[P, R \triangleright M, T]^{I,O}_{p,t} \to [P, R \triangleright M, T]^{I,O}_{p',t'} \qquad \frac{M'(l) = (\textit{self } \vec{x})P' \qquad \text{noEvent}(T,t)}{[\langle M'.l(\vec{v})\rangle, R \triangleright M, T]^{I,O}_{p,t} \to [\langle P'[M' \ \vec{v}/\textit{self } \vec{x}]\rangle, R \triangleright M, T]^{I,O}_{p,t'}} \quad \text{(R-MOVE,R-CALL)}$$

$$\frac{T' = T \uplus (l(\vec{v}), v_1, t + v_1, t + v_2) \qquad \text{noEvent}(T,t)}{\begin{array}{l}[\langle \textbf{timer } l(\vec{v}) \textbf{ every } v_1 \textbf{ expire } v_2\rangle, R \triangleright M, T]^{I,O}_{p,t} \to \\ \qquad [\langle \{\}\rangle, R \triangleright M, T']^{I,O}_{p,t'}\end{array}} \qquad \frac{t_1 \leq t_2 \qquad T' = T \uplus (l(\vec{v}), v, t_1 + v, t_0)}{\begin{array}{l}[P, R \triangleright M, T \uplus (l(\vec{v}), v, t_1, t_0)]^{I,O}_{p,t_2} \to \\ \qquad [P, R :: \textbf{let } x = M.l(\vec{v}) \textbf{ in } x \triangleright M, T']^{I,O}_{p,t_2}\end{array}}$$

(R-TIMER,R-INTERRUPT)

$$\frac{t_1 > t_2 > t_0}{[P, R \triangleright M, T \uplus (l(\vec{v}), v, t_1, t_0)]^{I,O}_{p,t_2} \to [P, R \triangleright M, T]^{I,O}_{p,t_2}} \quad \text{(R-EXPIRE)}$$

with $\langle P'\rangle$ an abbreviation for $\textbf{let } x = P' \textbf{ in } P$.

Fig. 4. Reduction semantics for sensors.

$$\frac{S \to S'}{S \mid S'' \to S' \mid S''} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \to S_3 \qquad S_3 \equiv S_4}{S_1 \to S_4} \qquad [P, R \triangleright M, T]^{I,m::O}_{p,t}\{S\} \to [P, R \triangleright M, T]^{I,O}_{p,t} \mid S$$

(R-NETWORK, R-CONGR, R-RELEASE)

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

Fig. 5. Reduction semantics for sensor networks.

simply logs the arguments using a built-in external call. Then, the program starts a timer for the receiver function with a period of 5 milliseconds for 10 seconds. Finally, the sink broadcasts a setup message with a period of 100 milliseconds and a duration of 10 seconds. The call is placed in the outgoing queue of the sink and eventually finds its way into the network.

```
// sink
install {
   receiver = (self)      receive
   gather   = (self,t,v) extern log(t,v) };
timer receiver() every 5 expire 10000;
send setup(100,10000)
```

```
// node
install {
   receiver = (self)
      receive
   setup  = (self,p,dt)
      timer self.sample() every p expire dt
   sample = (self)
      let t = extern time() in
      let v = extern data() in
      send gather(t,v)              };
timer receiver() every 5 expire 10000;
```

Each *node* starts by installing a module with a receiver function, similar to that on the sink, and a setup and a sample functions. Then, the node starts a timer on receiver and waits for incoming messages. When a sensor receives a setup message from the network, it creates a second timer to periodically call sample in the same module. When this function is executed the local clock and the desired data are read with external calls and a gather message is sent to the network carrying those values.

Note that the routing of messages is transparent at this level. It is controlled at the network and data-link layers and we model this by having an extra semantic layer for the network (*c.f.* Fig. 5). In this example, the messages from the sink are delivered to every sensor that carries a setup function. The information originating in the nodes, in the form of gather messages, on the other hand, is successively relayed up to the sink (since sensors have no gather functions implemented).

Streaming data II. This example illustrates the same application as above, but now the code for the nodes is sent over a wireless channel by the sink and installed dynamically. The nodes just have a very short listener installed. The function deploy receives a module and installs it. Together with a timer to receive messages from the network, this is the only basic code that sensor devices require for an application to be deployed and executed. The sink may, as in this case, use deploy messages to send the code to the devices and then issue a message to trigger the execution, in this case setup.

```
// sink
install {
   receiver = (self) receive
   gather   = (self,t,v) extern log(t,v) };
send deploy( {
   receiver = (self)
      receive
   setup  = (self,p,dt)
      timer self.sample() every p expire dt
   sample = (self)
      let t = extern time() in
      let v = extern data() in
      send gather(t,v)              })
timer receiver() every 5 expire 10000;
send setup(100,10000)

// node
install {
   deploy   = (self, m) install(m)
   receiver = (self) receive        };
timer receiver() every 5 expire 10000;
```

III. THE VIRTUAL MACHINE SPECIFICATION

This section presents a full specification for a virtual machine for the Callas programming model. The machine executes byte-code generated by the Callas compiler, the run-time structure of which may be seen in Fig. 6. Besides the basic language data types: booleans (b, 1 byte), integers (n, 4 bytes) and floating-point values (f, 4 bytes), the byte-code uses short integers (k, 1 byte) for the instruction opcodes and some instruction arguments.

A program $\mathcal{P}$ in byte-code format is composed of an integer value, the *magic number*, and an array of modules. The magic number is used for marshalling messages accross distinct device micro-processor architectures. Each

$$
\begin{array}{lll}
\textit{program} & \mathcal{P} \in & \textsc{Int} \times \textsc{ArrayOf}(\mathcal{M}) \\
\textit{module} & \mathcal{M} \in & \textsc{MapOf}(\textsc{String} \mapsto \textsc{ArrayOf}(v) \times \mathcal{B} \times \mathcal{U}) \\
\textit{bytes} & \mathcal{B} \in & \textsc{ArrayOf}(c) \\
\textit{symbols} & \mathcal{U} \in & \textsc{ArrayOf}(v) \\
\textit{instruction} & c \in \{ & \mathsf{update}, \mathsf{extern}, \mathsf{call}, \mathsf{timer}, \mathsf{return}, \mathsf{jmp}\ n, \mathsf{iftrue}\ n, \mathsf{receive}, \mathsf{send}, \\
 & & \mathsf{loadb}, \mathsf{storeb}, \mathsf{loadm}\ k, \mathsf{loadc}\ k, \mathsf{load}\ k, \mathsf{store}\ k, \mathsf{push}, \mathsf{pop}, \\
 & & \mathsf{add}, \mathsf{sub}, \mathsf{mul}, \mathsf{div}, \mathsf{not}, \mathsf{and}, \mathsf{eq}, \mathsf{lt}\ \} \\
\textit{value} & v \in & \textsc{Bool} \cup \textsc{Int} \cup \textsc{Float} \cup \mathcal{M}
\end{array}
$$

Fig. 6. The byte-code format.

$$
\begin{array}{lll}
\textit{machine state} & & \textsc{Int} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{I} \times \mathcal{O} \\
\textit{timers} & \mathcal{T} \in & \textsc{SetOf}(\mathcal{S} \times \textsc{Int} \times \textsc{Int} \times \textsc{Int}) \\
\textit{call-stack} & \mathcal{C} \in & \textsc{StackOf}(\textsc{Int} \times \langle \vec{v} \rangle \times \mathcal{S} \times \mathcal{B} \times \mathcal{U}) \\
\textit{run-queue} & \mathcal{R} \in & \textsc{QueueOf}(\mathcal{S} \times \mathcal{B}) \\
\textit{incoming-queue} & \mathcal{I} \in & \textsc{QueueOf}(\langle \vec{v} \rangle) \\
\textit{outgoing-queue} & \mathcal{O} \in & \textsc{QueueOf}(\langle \vec{v} \rangle) \\
\textit{operand-stack} & \mathcal{S} \in & \textsc{StackOf}(v)
\end{array}
$$

Fig. 7. The syntactic categories of the virtual machine.

module $\mathcal{M}$ is a map from strings (the function names) onto tuples composed of an array of values (the local variables for the function), a byte-code array ($\mathcal{B}$, the code), and a symbol array ($\mathcal{U}$, constants referred to in the byte-code). The latter is reminiscent of the Java Virtual Machine's [7] constant pool, and allows for more compact byte-code and a simpler instruction set. Each instruction in a byte-array is composed of an opcode, eventually followed by an integer argument (1 byte or 4 bytes). The instruction set is very simple with instructions for: manipulating modules, making calls, moving data, network I/O, control-flow and basic arithmetic, and logic operations. The symbol array, $\mathcal{U}$, is simply an array of constants of the basic types.

The virtual machine manipulates the same values as the source language, namely the basic types plus *modules*. Values are typically grouped in *frames*, representing messages or environments, and denoted $\langle \vec{v} \rangle$. Empty frames with k slots are denoted $\langle \mathring{k} \rangle$. A virtual machine state is represented by the tuple $\textsc{Int} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{I} \times \mathcal{O}$. The integer is the machine internal clock and uses arbitrary time units. The module $\mathcal{M}$ contains all the functions that have been installed in the sensor. The set $\mathcal{T}$ of timed-calls contains tuples with an operand-stack, holding the environment for a periodic call, and three integers, representing the period of the call, the time of the next call and the expiration time. Processes are executed in a call-stack $\mathcal{C}$. The latter manages call-frames, composed of a program counter, a data-frame for the process environment, an operand-stack $\mathcal{S}$, a byte-array $\mathcal{B}$ and a symbol array $\mathcal{U}$. The run-queue, $\mathcal{R}$, is a queue of pending processes. Each pending process is composed of an operand-stack $\mathcal{S}$ and a byte-array $\mathcal{B}$. Finally, the incoming and outgoing queues, $\mathcal{I}$ and $\mathcal{O}$ respectively, are used by the machine to interact with the network. They hold data-frames that encode calls. The items in stacks and queues of a syntactic category α are written: $\alpha_1 : \cdots : \alpha_n$ and $\alpha_1 :: \cdots :: \alpha_n$, respectively. Empty stacks and queues are denoted ϵ. A summary of the components of the virtual machine is given in Fig. 7.

A. Semantics

The virtual machine starts to execute when a byte-array $\mathcal{B}$ for the first function in a Callas program is fed into it. This function is assumed to be "run" from the module at offset 0. This function is assumed to have no parameters and, obviously, no free variables. Initially, the incoming-, outgoing-, and run-queues are empty. The set of timed calls is also empty, as is also the module containing the code installed in the sensor device. The call-stack, on the other hand, contains a tuple with a zero program counter, an empty environment data-frame, an empty operand-stack and, the byte-code and symbols corresponding to the "run" function. We write the initial state with our notation

as follows:

$$\langle 0, \emptyset, \emptyset, (0, \langle \mathring{k} \rangle, \epsilon, \mathcal{B}, \mathcal{U}), \epsilon \rangle_{\epsilon}^{\epsilon}$$

where,

$$(\langle \mathring{k} \rangle, \mathcal{B}, \mathcal{U}) \;=\; \mathcal{P}.getModule(0).getFunction(\text{“run”})$$

and k is the number of local variables for the function. After the first module is loaded, the execution of the program proceeds through a series of state transitions, designed to match the operational semantics given in Section II. The transition rules for the virtual machine can be seen in Fig. 8.

The update instruction takes two modules at the top of the operand-stack ($\mathcal{S}$) and merges them, the new module being placed at the top of the operand-stack.

The call instruction expects a module $\mathcal{M}_1$, a string l and a sequence of k_1 arguments $\vec{v}_2$ at the top of the operand-stack. The module and the string are used to get a copy of the environment frame, the byte-code, and the symbols for the function to be called. The environment frame is divided into three parts: the first slots for the parameters; next come the slots for the free variables $\vec{v}_1$; and the slots for the local variables introduced with **let** statements. The notation $\langle \mathring{k_1}, \vec{v_1}, \mathring{k_2} \rangle$ means an environment frame with k_1 empty slots, followed by the values $\vec{v}_1$, and finally, k_2 empty slots. The arguments $\vec{v}_2$ are placed in the environment frame in the initial slots. The free variables are loaded into the environment for the function when the module is first loaded (see instruction loadm below). The resulting call-frame is placed in the call-stack C. Another type of call, to functions external to the program byte-code, is supported with the instruction extern. Such a call returns immediately with the result value at the top of the operand stack. The send instruction takes a function name and a sequence of values from the operand stack, serializes them into a network format message, and places the latter in the O queue. Its complementary instruction, receive, takes a network format message from the I queue, de-serializes it into a pending process, and adds an entry for it in run-queue ($\mathcal{R}$). Note the byte-code snippet associated with these processes. When activated it loads the module with the machine's installed code onto the operand-stack and calls function l in it.

The following three rules handle timed calls. The first one, for instruction timer, programs a timer to periodically call function l with arguments $\vec{v}$, every n_1 time units, for a time lapse of n_2 units. The second rule has no instruction associated. It just triggers a call to l every time a period has passed in the device's clock. This is done on a best effort basis. No guarantees are given that the call with be triggered at the exact time instant. The third rule expires the timer by checking whether the timer exceeded the programmed time lapse. The next three rules handle the return from functions and context switching. The execution of a function always ends with a return instruction. In the first rule, the return value is placed at the top of the operand-stack in the previous tuple of the call-stack. The next rule handles the case where the running process exits and a pending process is fetched from the run-queue R. The final rule in this set checks whether the run-queue is also empty, in which case the machine loops until some (timer activated) process appears.

The next set of instructions handle data movement to and from the top of the operand-stack. For example, loadb (storeb) are used to load (store) a module from (to) the installed code module $\mathcal{M}$ of the device. The instruction loadm k' loads a generic module directly from the program byte-code where k' represents the index of the module in the program $\mathcal{P}$. The free variables for each of the k functions in the module are loaded from the operand stack into their respective environment frames by this instruction. Moving values between the environment data-frame and the operand-stack is done using load k and store k. Constants referenced in the byte-code are kept in a symbol array on a per function basis. The constants may be loaded with loadc k, where k is the offset of the symbol in the symbol array.

The remainder of the instructions are rather standard. Their definition is based on similar instructions in the Java virtual machine.

IV. SOUNDNESS OF THE VIRTUAL MACHINE

In this section we present the main result of this paper that connects the operational semantics of the programming language with that of the virtual machine. To do so, we need to translate programs in the source language into equivalent byte-code programs that can be executed by the virtual machine. The format of the byte-code is given in Fig. 6.

$$\frac{\mathcal{B}[j] = \textsf{update}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{M}_1 : \mathcal{M}_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \mathcal{M}_1 + \mathcal{M}_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{call} \qquad (\langle \mathring{k_1}, \vec{v_1}, \mathring{k_2}\rangle, \mathcal{B}', \mathcal{U}') = \mathcal{M}_1.getFunction(l)}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{M}_1 : l : k_1 : \vec{v}_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (0, \langle \vec{v}_2, \vec{v}_1, \mathring{k_2}\rangle, \epsilon, \mathcal{B}', \mathcal{U}') : (j+1, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{extern} \qquad v = callExtern(l : k : \vec{v})}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, l : k : \vec{v} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{send} \qquad \Lambda = \mathcal{P}.getMagic() \qquad \langle \Lambda, l, k, \vec{v}\rangle = pack(\Lambda, l : k : \vec{v})}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, l : k : \vec{v} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}::\langle \Lambda, l, k, \vec{v}\rangle}}$$

$$\frac{\mathcal{B}[j] = \textsf{receive} \qquad \Lambda_1 = \mathcal{P}.getMagic() \qquad S = unpack(\Lambda_1, \langle \Lambda_2, l, k, \vec{v}\rangle)}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \_, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\langle \Lambda_2, l, k, \vec{v}\rangle :: \mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \_, \mathcal{B}, \mathcal{U}) : \mathcal{C}, R :: (S, [\textsf{loadb}, \textsf{call}])\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{timer} \qquad S' = l : k : \vec{v}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, n_1 : n_2 : S' : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T} + \{(S', n_1, t + n_1, t + n_2)\}, (j+1, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{t_1 \leq t \qquad S = l : k : \vec{v}}{\langle t, \mathcal{M}, \mathcal{T}.\{(S, n, t_1, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t, \mathcal{M}, \mathcal{T}.\{(S, n, t_1 + n, t_2)\}, \mathcal{C}, (S, [\textsf{loadb}, \textsf{call}]) :: \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{t > t_2}{\langle t, \mathcal{M}, \mathcal{T}.\{(S, \_, \_, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{return}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, v, \mathcal{B}, \mathcal{U}) : (\_, \_, S', \_, \_) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (\_, \_, v : S', \_, \_) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\langle t, \mathcal{M}, \mathcal{T}, \epsilon, (l : k : \vec{v}, [\textsf{loadb}, \textsf{call}]) :: R\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (0, \epsilon, l : k : \vec{v}, [\textsf{loadb}, \textsf{call}]), \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}$$

$$\langle t, \mathcal{M}, \mathcal{T}, \epsilon, \epsilon\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, \epsilon, \epsilon\rangle^{\mathcal{I}}_{\mathcal{O}}$$

$$\frac{\mathcal{B}[j] = \textsf{loadb}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \mathcal{M} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{storeb}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{M}_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}_1, \mathcal{T}, (j+1, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{loadm } k' \qquad (k, \mathcal{M}_1) = \mathcal{P}.getModule(k')}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, n_1 : m_1 : \vec{v_1} : \cdots : n_k : m_k : \vec{v_k} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \mathcal{M}_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{loadc } k \qquad v = \mathcal{U}[k]}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{load } k \qquad v = \langle \vec{v}\rangle[k]}{\langle t, \mathcal{M}, \mathcal{T}, (j, \langle \vec{v}\rangle, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \langle \vec{v}\rangle, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{store } k \qquad \langle \vec{v}\rangle[k] = v}{\langle t, \mathcal{M}, \mathcal{T}, (j, \langle \vec{v}\rangle, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \langle \vec{v}\rangle, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{push } k}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, k : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{pop}}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = binop \qquad v = binop(v_1, v_2)}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, v_1 : v_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = unop \qquad v = unop(v_1)}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, v_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+1, \_, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{jmp } n}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+n, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{iftrue } n}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \textsf{True} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+n, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

$$\frac{\mathcal{B}[j] = \textsf{iftrue } n}{\langle t, \mathcal{M}, \mathcal{T}, (j, \_, \textsf{False} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \to \langle t', \mathcal{M}, \mathcal{T}, (j+5, \_, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}}$$

Fig. 8. Reduction semantics for the virtual machine.

$$[\![P]\!] \stackrel{\text{def}}{=} (\Lambda, \langle m_1 \ldots m_n \rangle)$$
where $m_i = \mathcal{M}[\![M_i]\!](M_1 \ldots M_n)$ and $M_i \in \text{modules}(\{run = ()P\}) \cup \{\{\}\}$

Fig. 9. The translation of Callas programs.

$$\begin{aligned}
&\text{modules}(i) = \text{modules}(f) = \text{modules}(b) = \text{modules}(x) = \emptyset \\
&\text{modules}(v_1 \ldots v_n) = \bigcup_{i=1..n} \text{modules}(v_i) \\
&\text{modules}(e_1 \; op \; e_2) = \text{modules}(e_1) \cup \text{modules}(e_2) \\
&\text{modules}(\textbf{not } e) = \text{modules}(e) \\
&\text{modules}(\{l_i = (\vec{x})P_i\}_{i \in I}) = \; \{\{l_i = (\vec{x})P_i\}_{i \in I}\} \cup \bigcup_{i \in I} \text{modules}(P_i) \\
&\text{modules}(v.l(\vec{v})) = \; \text{modules}(v\vec{v}) \\
&\text{modules}(\textbf{load}) = \; \emptyset \\
&\text{modules}(\textbf{store } v) = \; \text{modules}(v) \\
&\text{modules}(\textbf{extern } l(\vec{v})) = \; \text{modules}(\vec{v}) \\
&\text{modules}(\textbf{timer } l(\vec{v}) \textbf{ every } v \textbf{ expire } v') = \; \text{modules}(\vec{v}) \\
&\text{modules}(\textbf{send } l(\vec{v})) = \; \text{modules}(\vec{v}) \\
&\text{modules}(\textbf{receive}) = \; \emptyset \\
&\text{modules}(\textbf{let } _ = P_1 \textbf{ in } P_2) = \; \text{modules}(P_1) \cup \text{modules}(P_2) \\
&\text{modules}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) = \; \text{modules}(P_1) \cup \text{modules}(P_2)
\end{aligned}$$

Fig. 10. Function modules.

Fig. 9 shows the top-level translation of a program P. The latter is translated into a pair that contains a magic number Λ, to be used in messaging, and an array of modules in byte-code format. These are the translations of *all* the modules that occur statically in the source code for the program. The modules are extracted by calling the auxiliary function $modules$. Note that we embed the top-level program P in a module containing a single function run whose body is the code for P.

The function modules, Fig. 10, is straightforward. It simply transverses the program structure and collects all the modules it finds. For example, when applied to a module $\{l_i = (\vec{x}_i)\, P_i\}_{i \in I}$, it collects the module itself and then recursively collects all the modules that may exist in the bodies P_i of the functions in the module.

Fig. 11 presents the translations for modules and for the functions within. Again matching the byte-code format, a module is translated into a map from strings (the function names) into the translations of functions. Each function is translated into a triple that contains: an array of empty slots, for the environment, with size equal to the total number of variables used in the function (parameters, free variables, and variables introduced with **let** statements); an array with the byte-code instructions; and an array with the literals that occur in the source code for the function (*i.e.* numeric constants and strings). The constants in the source code of functions are collected using the auxiliary function $consts$, that recursively descends the structure of the source code for the body of the function and collects any constants it finds. For example, in a call $v.l(\vec{v})$, it collects the function name l and inspects the target of the call and arguments for more constants.

Finally, we have the translations for values (Fig. 13) and for the language constructs (Fig. 14). The translation function for both carries an environment that contains information on all the variables used by the function ($\vec{y}$), all the modules known to the program $\vec{M}$ and, the constants used by the function ($\vec{b}$).

A constant is translated as a loadc instruction with an argument that is the offset of the constant in the constant pool for the function. The case for variables is similar except that load is used with an argument that corresponds to the offset for the variable in the function's environment ($\vec{y}$).

$$\mathcal{M}[\![\{l_i = (\vec{x})P_i\}_{i\in I}]\!](\vec{M}) \stackrel{\text{def}}{=} \{l_1 \to f_1, \ldots, l_n \to f_n\}$$
$$\text{where } f_i = \mathcal{F}[\![l_i = (\vec{x}_i)P_i]\!](\vec{M})$$

$$\mathcal{F}[\![l = (\vec{x})P]\!](\vec{M}) \stackrel{\text{def}}{=} (\langle \mathring{k} \rangle, \langle \vec{c} \rangle, \langle \vec{b} \rangle)$$
$$\text{where } \vec{c} = \mathcal{P}[\![P]\!](\{\vec{x}\} \cup \text{fn}((\vec{x})P), \vec{M}, \vec{b}); \texttt{return}$$
$$k = |\text{bn}((\vec{x})P) + \text{bn}((\vec{x})P)|$$
$$\vec{b} = \text{consts}(P)$$

Fig. 11. Translation of modules.

$$\text{consts}(i) = \{i\} \qquad \text{consts}(f) = \{f\} \qquad \text{consts}(b) = \{b\} \qquad \text{consts}(x) = \emptyset$$
$$\text{consts}(v_1 \ldots v_n) = \bigcup_{i=1..n} \text{consts}(v_i)$$
$$\text{consts}(e_1\ op\ e_2) = \text{consts}(e_1) \cup \text{consts}(e_2)$$
$$\text{consts}(\textbf{not}\ e) = \text{consts}(e)$$
$$\text{consts}(\{l_i = (\vec{x})P_i\}_{i\in I}) = \{l_i\}_{i\in I} \cup \bigcup_{i\in I} \text{consts}(P_i)$$
$$\text{consts}(v.l(\vec{v})) = \{l\} \cup \text{consts}(v\vec{v})$$
$$\text{consts}(\textbf{load}) = \emptyset$$
$$\text{consts}(\textbf{store}\ v) = \text{consts}(v)$$
$$\text{consts}(\textbf{extern}\ l(\vec{v})) = \{l\} \cup \text{consts}(\vec{v})$$
$$\text{consts}(\textbf{timer}\ l(\vec{v})\ \textbf{every}\ v\ \textbf{expire}\ v') = \{l\} \cup \text{consts}(\vec{v}vv')$$
$$\text{consts}(\textbf{send}\ l(\vec{v})) = \{l\} \cup \text{consts}(\vec{v})$$
$$\text{consts}(\textbf{receive}) = \emptyset$$
$$\text{consts}(\textbf{let}\ \_ = P_1\ \textbf{in}\ P_2) = \text{consts}(P_1) \cup \text{consts}(P_2)$$
$$\text{consts}(\textbf{if}\ b\ \textbf{then}\ P_1\ \textbf{else}\ P_2) = \text{consts}(b) \cup \text{consts}(P_1) \cup \text{consts}(P_2)$$

Fig. 12. Function consts.

The most elaborate case is that of loading modules. In this case the instruction loadm is used but only after the values for the free variables for each of the functions in the module are loaded into the operand stack by a succession of loadc and load instructions. These values will then be copied internally by loadm into the environments of each of the functions of the module. For example, in Fig. 13, the values for the free variables for function l_1 are translated by $\mathcal{C}[\![l_1(|\vec{x}_1|, \text{fn}((\vec{x}_1)P_1))]\!](\vec{y}, \vec{M}, \vec{u})$, which also generates an instruction to push the number of free variables (k') and the offset in the function environment where they should be positioned (k) into the operand stack.

The translation of programs, Fig. 14, is rather straightforward and compact. We describe just a few cases. A call $v.l(\vec{v})$ is translated into the translation of its arguments and function name ($\mathcal{A}[\![l(\vec{v})]\!](\vec{y}, \vec{M}, \vec{u})$), the translation of the target module ($\mathcal{P}[\![v]\!](\vec{y}, \vec{M}, \vec{u})$) and the instruction `call`. The translation of the arguments and function name involves translating each argument in turn, pushing the number of arguments and translating the function name. The case for **extern** $l(\vec{v})$ is similar except that we do not need to translate the target (this is a direct system call). The construct **timer** $l(\vec{v})$ **every** v **expire** v' has a slightly more contrived translation. First the arguments and function name for the call are translated, then the period of the call and the expiration time, then a `timer` instruction that will program the timer, and finally, the empty module as the result of the operation. The other cases are fairly straightforward.

$$\mathcal{P}[\![v]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \texttt{loadc}\ i, \text{where}\ v = \vec{u}[i]$$
$$\mathcal{P}[\![y_i]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \texttt{load}\ i$$
$$\mathcal{P}[\![\{l_i = (\vec{x}_i)P_i\}_{i\in I}]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=}$$
$$\quad \mathcal{C}[\![l_1(|\vec{x}_1|, \text{fn}((\vec{x}_1)P_1))]\!](\vec{y},\vec{M},\vec{u}); \ldots; \mathcal{C}[\![l_n(|\vec{x}_n|, \text{fn}((\vec{x}_n)P_n))]\!](\vec{y},\vec{M},\vec{u}); \texttt{loadm}\ j$$
$$\text{where}\ \vec{M}\ \text{is}\ M_1 \cdots \underbrace{\{l_i = (\vec{x}_i)P_i\}_{i\in I}}_{M_j} \cdots M_m$$
$$\mathcal{C}[\![l(k', v_k \cdots v_1)]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{P}[\![v_1]\!](\vec{y},\vec{M},\vec{u}); \cdots; \mathcal{P}[\![v_k]\!](\vec{y},\vec{M},\vec{u}); \texttt{push}\ k; \texttt{push}\ k'$$

Fig. 13. Translation of values and of sequences of values.

$$\mathcal{P}[\![\textbf{load}]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \texttt{loadb}$$
$$\mathcal{P}[\![\textbf{store}\ v]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{P}[\![v]\!](\vec{y},\vec{M},\vec{u}); \texttt{storeb}; \mathcal{P}[\![\{\}]\!](\vec{y},\vec{M},\vec{u})$$
$$\mathcal{P}[\![v.l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{A}[\![l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}); \mathcal{P}[\![v]\!](\vec{y},\vec{M},\vec{u}); \texttt{call}$$
$$\mathcal{P}[\![\textbf{extern}\ l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{A}[\![l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}); \texttt{extern}$$
$$\mathcal{P}[\![\textbf{timer}\ l(\vec{v})\ \textbf{every}\ v\ \textbf{expire}\ v']\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=}$$
$$\quad \mathcal{A}[\![l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}); \mathcal{P}[\![v']\!](\vec{y},\vec{M},\vec{u}); \mathcal{P}[\![v]\!](\vec{y},\vec{M},\vec{u}); \texttt{timer}; \mathcal{P}[\![\{\}]\!](\vec{y},\vec{M},\vec{u})$$
$$\mathcal{P}[\![\textbf{send}\ l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{P}[\![l(\vec{v})]\!](\vec{y},\vec{M},\vec{u}); \texttt{send}; \mathcal{P}[\![\{\}]\!](\vec{y},\vec{M},\vec{u})$$
$$\mathcal{P}[\![\textbf{receive}]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \texttt{receive}; \mathcal{P}[\![\{\}]\!](\vec{y},\vec{M},\vec{u})$$
$$\mathcal{P}[\![\textbf{let}\ x_i = P_1\ \textbf{in}\ P_2]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{P}[\![P_1]\!](\vec{y},\vec{M},\vec{u}); \texttt{store}\ i; \mathcal{P}[\![P_2]\!](\vec{y}x_i,\vec{M},\vec{u})$$
$$\mathcal{P}[\![\textbf{if}\ b\ \textbf{then}\ P_1\ \textbf{else}\ P_2]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=}$$
$$\quad \mathcal{P}[\![b]\!](\vec{y},\vec{M},\vec{u}); \texttt{iftrue}\ i_2; \mathcal{P}[\![P_2]\!](\vec{y},\vec{M},\vec{u}); \texttt{jmp}\ i_1; \mathcal{P}[\![P_1]\!](\vec{y},\vec{M},\vec{u})$$
$$\quad \text{where}\ i_1 = |\mathcal{P}[\![P_1]\!](\vec{y},\vec{M},\vec{u})|$$
$$\quad\quad \text{and}\ i_2 = |\mathcal{P}[\![P_2]\!](\vec{y},\vec{M},\vec{u})|$$
$$\mathcal{A}[\![l(v_k \cdots v_1)]\!](\vec{y},\vec{M},\vec{u}) \stackrel{\text{def}}{=} \mathcal{P}[\![v_1]\!](\vec{y},\vec{M},\vec{u}); \cdots; \mathcal{P}[\![v_k]\!](\vec{y},\vec{M},\vec{u}); \texttt{push}\ k; \mathcal{P}[\![l]\!](\vec{y},\vec{M},\vec{u})$$

Fig. 14. Translation of programs.

The main result of this paper is that the semantics of Callas programs. In other words, assume we start with a byte-code program $[\![P]\!]$ that is the translation of a Callas program P, and that this program, as it is executed by the virtual machine, evolves through a sequence of transitions into another program $[\![P']\!]$, which is again the translation of some Callas program P'. We show that there exists a one step reduction using the Callas semantics from P to P'. The following diagram informally illustrates our description.

$$\begin{array}{ccc} P & \longrightarrow & P' \\ \downarrow & & \downarrow \\ [\![P]\!] & \longrightarrow^{\star} & [\![P']\!] \end{array}$$

Formally, this result could be stated as the following

Theorem 1. *if* $[\![P]\!] \rightarrow^{*} [\![P']\!]$*, then* $P \rightarrow P'$

Proof: (sketch) We take P and proceed by induction on the translation of P, analysing the last rule applied from the translation function (*vide* Fig. 14). For each case we consider the result of translating P into byte-code program $[\![P]\!]$; next we use the semantic rules from Fig. 8 to compute $[\![P']\!]$ such that it is exactly the result of

translating Callas program P'. Using the rules in Fig. 4 we prove each case, namely that $P \to P'$. For the cases of the **let** $x = P_1$ **in** P_2 and **if** b **then** P_1 **else** P_2 instructions we need to use the induction hypothesis, since for such cases the function is used recursively to translate programs P_1 and P_2.

V. CONCLUSIONS

In this paper we presented a virtual machine specification for the Callas programming language for sensor networks and proved its soundness. In other words, we showed that byte-code obtained by translating Callas programs is executed by sequences of transitions of the virtual machine that are emulated by the semantics of the programming language.

This result, coupled with the type-safety of the programming language [11], firmly establishes the robustness of the framework for programming sensor networks. In fact, not only can we filter out type-unsafe programs at compile time, but we also ensure, up to some implementation error of the virtual machine, that, at run-time, programs preserve the programming language semantics. We have implemented a Callas language compiler and a prototype of the virtual machine for the SunSPOT platform that will allow us to experiment with the language and perform large scale simulations in future work.

We argue that this approach for developing programming languages for sensor networks is relevant since it allows the premature (static) detection of would-be run-time errors in programs and minimizes the risk of program misbehavior or failure at run-time, a situation that is highly problematic in the context of sensor networks given the characteristics of the deployments (e.g. physical access to the nodes impossible, unreliable communication links, etc.).

Acknowledgments. This work was sponsored by Fundação para a Ciência e Tecnologia under project CALLAS (contract PTDC/EIA/71462/2006).

REFERENCES

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

[2] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modelling of Sensor Nets in Ptolemy II. In *Proceedings of the Third International Symposium on Information Processing in Sensor Networks (IPSN'04)*, pages 359–368. ACM Press, 2004.

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

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

[5] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'91)*, number 512 in LNCS, pages 133–147. Springer-Verlag, 1991.

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

[7] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

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

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

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

[11] F. Martins, L. Lopes, and J. Barros. Towards Safe Programming of Wireless Sensor Networks. In *Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'09), part of ETAPS'09, York*, 2009.

[12] N. Mezzetti and D. Sangiorgi. Towards a Calculus for Wireless Systems. In *MFPS'06*, volume 158 of *ENTCS*, pages 331–354. Elsevier Science, 2006.

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

[14] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[15] R. Newton, Arvind, and 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)*, pages 37–44, 2005.

[16] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'06)*. ACM Press, June 2006.