Safe-By-Design Programming Languages for Wireless Sensor Networks

Tiago Cogumbreiro[1], Pedro Gomes[2]
Francisco Martins[1], Luís Lopes[2]

[1]LASIGE, Faculdade de Ciências
Universidade de Lisboa

[2]CRACS & INESC-Porto LA, Faculdade de Ciências
Universidade do Porto

Technical Report Series: DCC-2011-09

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/

Safe-By-Design Programming Languages for Wireless Sensor Networks

Tiago Cogumbreiro
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Email: cogumbreiro@di.fc.ul.pt
Pedro Gomes
CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto
Email: pedro.gomes@dcc.fc.up.pt
Francisco Martins
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Email: fmartins@di.fc.ul.pt
Luís Lopes
CRACS & INESC-Porto LA, Faculdade de Ciências, Universidade do Porto
Email: lblopes@dcc.fc.up.pt

June 8, 2011

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 that targets these platforms. In particular, current systems do not place enough stress on providing formal descriptions of the language and its run-time system, and on proving important static properties. As a contrasting approach, we design, implement, and deploy a programming language along with a run-time system that enable us to prove two fundamental static properties: the type-safety of the language and the soundness of its run-time system. These properties ensure the absense of an important class of run-time errors in sensor network applications, which shortens development time and simplifies debugging.

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 infrastructure 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 (e.g. MANETs). The sensor devices are extremely limited in terms of hardware (CPU, memory) and power resources (typically batteries). Their deployment at remote locations makes physical access to the devices, *e.g.*, for maintenance and debugging, difficult if not impossible.

There are many proposals for programming languages for WSN providing the programmers with distinct levels of hardware and network awareness and distinct programming abstractions [8]. For example, at the very lowest level, running on the bare hardware, we have Pushpin [7]. Abstracting away from the hardware there are languages like the (ubiquitous) component-based language nesC [4]. Higher up in the abstraction level we find macroprogramming languages that allow programmers to abstract away, not only from sensor devices, but also from the network infra-structure, by resorting to sophisticated compilers. They provide abstractions such as streams, in Regiment [13], databases, in TinyDB [10], and agents, in SensorWare [3].

Despite this diversity of proposals, current programming languages for WSN are quite vulnerable to errors and this has negative impact on the usability of the platforms. 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 byte-code). Macroprogramming languages are illustrative of this state of affairs. Regiment [13], for example, a strongly typed functional macroprogramming language, is compiled into a low-level token machine language, which is then itself compiled into a nesC implementation of the run-time based on the distributed token machine model. Building a programming language in this way makes it rather difficult to establish a link between the semantics of the language and that of the corresponding run-time. Moreover, there is often no formal definition of the language upon which one could prove static and run-time properties of programs.

Run-time errors in sensor network applications can have multiple origins: (a) they can be due to device malfunction or interference from the environment; (b) the application may not work correctly due to semantic errors introduced by the programmer; (c) the system executing the program does not behave as expected, and; (d) the compiler is generating an incorrect program.

Errors of the first type are difficult or impossible to eliminate in most deployments. In general, this would involve physical access to the devices, which is not practical or even possible. The second type of error can be controlled by imposing an adequate programming discipline, *e.g.*, types, and by carefully testing the application before deployment. The third and fourth types are far more subtle but very important, as they may undermine a deployment with seemingly unexplainable errors and result in significant extra costs.

In this paper we argue that the errors of types (b), (c), and (d) can be minimised or even eliminated from sensor network applications by carefully designing the programming languages and the corresponding run-time systems. In particular, a programming language and its run-time system should feature two fundamental properties, respectively: type-safety and soundness. Language type-safety ensures that well-typed programs do not give rise to run-time protocol errors. A compiler for a type-safe programming language can statically type-check code and identify would-be run-time protocol errors, before the application is deployed over the network. This addresses errors of type (b). On the other hand, the soundness property ensures that the underlying run-time system preserves the semantics of the programming language. This is achieved by implementing the run-time system based on an abstract specification (e.g. a virtual machine) that can be proved to preserve the semantics of the programming language. This still leaves some margin for errors introduced by the programmer of the run-time, but these can be mitigated through an extensive evaluation and testing of the software. This addresses errors of type (c). As for errors of type (d), these can also be partially eliminated by proving that the language compiler preserves the semantics of the programming language when translating from source to executable code. This, however, will not be addressed in this paper and 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 [9], a programming language for WSN. The language derives from a core-calculus with precise static and operational semantics, based on the formalism of process calculi [6, 12]. For this language we have proved type-safety [11]. Here, we present a virtual machine specification for the Callas run-time and prove that it preserves the operational semantics of the language. With this semantically robust framework in the background we developed a prototype for the language compiler and another for the run-time system, whose architecture and implementation we describe.

Thus, in summary, the main contributions of this paper are:

- a methodology for the development of programming languages for WSN that are free of a large class of run-time errors by design;

- a specification for the Callas run-time system and the proof that it preserves the language semantics;

- a complete description of a proof-of-concept programming language, compiler, and run-time system that runs on the Sun SPOT platform [15] and on the VisualSense simulation tool [2].

The remainder of the paper is structured as follows. Section 2 presents the Callas language, the core-calculus, the operational semantics and, the type-safety result. Section 3 presents the syntax of the byte-code for the Callas virtual machine, the compilation scheme from Callas source programs into Callas byte-code, the virtual machine specification, and the run-time soundness result. Section 4 describes the architecture, and the implementation of the Callas compiler and run-time system. Finally, Section 5 ends the paper with some conclusions and perspectives for future work.

2 THE PROGRAMMING MODEL

This section aims at describing Callas, a programing language for sensor networks that offers constructs to describe sensor computations, communications, code mobility, and code updates. The language is based on a calculus [9, 11] with the goal of establishing a foundation for developing programming languages and run-time systems for sensor networks.

We start by presenting the language along with some examples 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). In Section 2.4 we state informally a type safety result — the interested reader may refer to [11] for the details.

2.1 The Callas Programming Language

We introduce the Callas language by example, programming a sensor node that periodically reads the ambient temperature and sends it to the network, as listed in Figure 1. A Callas program is a sequence of *terms*, whose components are type and module declarations, assignments, expressions, and conditionals. We adopt Python's line-oriented syntax, where *indentation* (the number of spaces in the beginning of a line) demarcates syntactic terms.

```
 1 defmodule Nil:
 2   pass
 3
 4 defmodule Sampler:
 5   Nil sample()                       # sample the temperature
 6 # declare module sampler, install it, and call sample() periodically
 7 module s of Sampler:
 8   def sample(self):
 9     curTemp = extern getTemp() # sense the temperature
10     send log(curTemp)
11
12 mem = load                 # load the sensor memory
13 newMem = mem || s          # update function sample
14 store newMem               # replace the sensor memory
15 # invoke sample() every ten minutes, for one week
16 sample() every 60*10 expire 60*60*24*7
```

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

The program starts with two type declarations (lines 1–2 and 4–5). The first type declaration begins with the reserved word **defmodule**, followed by a type identifier Nil (must be capitalised) 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 2, we define an empty module type (with zero functions). Keyword **pass** defines an empty sequence of syntactic terms, used to declare an empty syntactic block. In lines 4–5, we find the declaration of a type Sampler, a module with a function named sample that expects no arguments and returns empty modules. In lines 7–10, we declare a module, the first line holds the *module header* and then the *module body.* The module header begins with the reserved word **module**, succeeded by a variable s that binds the module—variables must begin with a lower-case letter—, followed by the reserved word **of**, then a type identifier Sampler 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 (in line 8) starts with the reserved word **def**, succeeded by the name of the function sample, and by one or more (comma-separated) parameters in parenthesis. The first parameter in any function is the module itself, *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 to a module. Notice that, as in Python, when a line ends with a colon the remaining lines are a syntactic group with an increased indentation.

The body of function sample consists of two terms. The first assigns to variable curTemp the result from an external call that gets the ambient temperature from 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 neighbouring nodes. This expression yields as a result an empty module (that is used as the outcome of function sample). There are no guaranties whether any sensor in the network picks up these remote function calls. The programmer must develop protocols for making sure messages are delivered.

The syntax of values comprises three categories: built-in values, operations (binary and unary), and variables. We adopt the Python's syntax for binary and unary operations as well as for built-in values.

The memory of a sensor may be replaced dynamically throughout the lifetime of the device. For accessing the memory of a device we use expressions **load** and **store**. In lines 12–14, we load the code of the device and save it in variable mem (line 12), assign to variable newMem a new module, by composing the modules in variables mem and s (line 13), and store the new module in memory (line 14). Expression $x \mid\mid y$ 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 name appearing on both modules). The syntax for operator $\mid\mid$ is based on the asymmetric merge operator of the record calculus [5].

Having an efficient power usage is essential when programming for WSN. The Callas programming language offers timed calls to allow the device to conserve energy between periodic computations, whenever possible. We program a timed-event in line 16 that invokes function sample, thus sending the sensed temperature to the network every ten minutes, for one week.

To conclude our first example, a network of devices executing the code in Figure 1 needs one or more devices that are programmed to receive the data and process it. Devices responsible for aggregating the results from the sensor network are usually called *sinks.* We list the Callas code in Figure 2 that records the maximum received temperature in the memory of the sink. Line 5 defines the type of function log with a typed parameter named temp of type **float**. Each typed parameter consists of a type and a name (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 (capitalised) type identifiers. The sink actually executes the code starting from line 11. First, we declare a module and assign it to variable sink. The declared module is an implementation of type Sink. 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 it is greater than the previous known maximum, the function updates the received temperature by storing module newMax, which updates function maxTemp to hold the new maximum sensed value. Notice that when the else branch is omitted, its value is the empty module. Function listen accepts a function call (for later execution) that arrives from the network if there is one. Expression **receive** evaluates to the empty module. We program a timed-event in line 30 that invokes function listen every 10 minutes, for one week. The programmer is responsible for handling if and when remote function calls get handled via expression **receive**.

Notice 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 neighbourhood. A receiving device captures the event and calls the corresponding call-back l that must reside in its memory.

The next example also illustrates an application where nodes periodically read and broadcast the ambient temperature, but now the code for the nodes is sent over a wireless channel by the sink and is installed dynamically. Callas supports *code mobility* by allowing sensors to communicate modules; the code "moves" physically in the network. We list the code for the sink in Figure 3, adapted from Figure 2. In lines 23–26, we declare a module named node. A device invoking function run of this module becomes configured as a node, since it executes the

```
defmodule Nil:
  pass

defmodule Sink:
  Nil log(float temp)
  Nil listen()

defmodule MaxTemp:
  float maxTemp()

module sink of Sink:
  def log(self, temp):
    mem = load
    maxTemp = mem.maxTemp()
    needsUpdate = temp > maxTemp
    if needsUpdate:
      module newMax of MaxTemp:
        def maxTemp(self):
          temp
      mem = mem || newMax
      store mem

  def listen(self):
    receive
# update sensor memory
mem = load
mem = mem || sink
store mem
# invoke listen() every ten minutes, for one week
listen() every 60*10 expire 60*60*24*7
```

Figure 2: A program for storing the maximum temperature received.

```
1  defmodule Nil:
2    #... (as in Figure 2)
3
4  defmodule Sink:
5    #... (as in Figure 2)
6
7  defmodule MaxTemp:
8    #... (as in Figure 2)
9
10 defmodule Sampler:
11   #... (as in Figure 1)
12
13 defmodule Runner:
14   Nil run()
15
16 module sink of Sink:
17   # ... (as in Figure 2)
18
19 # update sensor memory
20 # ... (as in Figure 2, lines 26–28)
21
22 # Function run() of this module executes the code in Figure 1
23 module node of Runner:
24   def run(self):
25     # declare module sampler, install it, and call sample() periodically
26     # ... (as in Figure 1, lines 7–16)
27
28 send deploy(node)
29 listen() every 60*10 expire 60*60*24*7
```

Figure 3: A program for storing the maximum temperature received.

code listed in Figure 1. Next, in line 28, we send module node to neighbouring peers, by using node as an argument in the remote function call of deploy. The nodes bootstrap with a short listener code, depicted in Figure 4. Function deploy receives and invokes function run, allowing remote code to perform any operation in the device. The sink can use deploy messages to deliver code to the devices in the network whenever needed.

The complete grammar for Callas is depicted in Figure 5.

2.2 Abstract Syntax

In order to define the semantics of our language we introduce an abstraction of the syntax described in Figure 5, by inserting a new **let** expression that handles the binding constructs uniformly (variable assignment and module definition), that makes explicit the scope of the bindings, and that enforces an evaluation order on expressions. Recall that in the concrete syntax of Callas, assignments and module definitions introduce new variables that are visible

```
defmodule Nil:
  pass

defmodule Runner:
  Nil run()

defmodule Node:
  Nil deploy(Runner code)
  Nil listen()

module node of Node:
  def deploy(self, code):
    # we execute the code received from the network
    code.run()
  def listen(self):
    receive

mem = load
newMem = mem || node
store newMem
# invoke sample() every ten minutes, for one week
listen() every 60*10 expire 60*60*24*7
```

Figure 4: A program for sending the sensed temperature over to the network periodically.

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

$$
\begin{array}{rcll}
e & ::= & & \textit{Expressions} \\
 & & v & \text{value} \\
 & | & \text{unop}\ v & \text{unary op.} \\
 & | & v\ \text{binop}\ v & \text{binary op.} \\
 & | & \textbf{\textsf{load}} & \text{load} \\
 & | & \textbf{\textsf{store}}\ v & \text{store} \\
 & | & v \mid\mid v & \text{merge modules} \\
 & | & v.l(\vec{v}) & \text{function call} \\
 & | & \textbf{\textsf{extern}}\ l(\vec{v}) & \text{external call} \\
 & | & l(\vec{v})\ \textbf{\textsf{every}}\ e\ \textbf{\textsf{expire}}\ e & \text{timed call} \\
 & | & \textbf{\textsf{send}}\ l(\vec{v}) & \text{communication} \\
 & | & \textbf{\textsf{receive}} & \text{communication} \\
v & ::= & & \textit{Values} \\
 & & x & \text{variable} \\
 & | & \ldots \mid 0 \mid \ldots & \text{integer} \\
 & | & \textbf{\textsf{True}} \mid \textbf{\textsf{False}} & \text{boolean} \\
 & | & \ldots \mid 0.0 \mid \ldots & \text{floating point}
\end{array}
$$

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

Figure 5: The syntax of Callas.

$$\begin{array}{llllll}
e ::= & \textit{Expressions} & M & ::= \{l_i(\vec{x}_i) = e_i\}_{i\in I} & \textit{Modules} \\
\quad \ldots & \text{same as in Figure 5} & v & ::= & \textit{Values} \\
\quad \mid \textbf{let}\ x = e\ \textbf{in}\ e & \text{sequence} & & \quad \ldots & \text{same as in Figure 5} \\
\quad \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional} & & \mid M & \text{modules}
\end{array}$$

Figure 6: The abstract syntax of Callas.

$$\begin{aligned}
[\![\vec{d}\ \vec{t}]\!] &= [\![\vec{t}]\!] \\
[\![\textbf{if}\ e : \P\ \vec{t_1}\ \textbf{else}\ : \P\ \vec{t_2}\ \vec{t_3}]\!] &= \textbf{let}\ x = \textbf{if}\ e\ \textbf{then}\ [\![\vec{t_1}]\!]\ \textbf{else}\ [\![\vec{t_2}]\!]\ \textbf{in}\ [\![\vec{t_3}]\!], x \notin \mathrm{fn}([\![\vec{t_3}]\!]) \\
[\![\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}\ \vec{t}]\!] &= \textbf{let}\ x = \{[\![\vec{f}]\!]\}\ \textbf{in}\ [\![\vec{t}]\!] \\
[\![x = e\ \P\ \vec{t}]\!] &= \textbf{let}\ x = e\ \textbf{in}\ [\![\vec{t}]\!] \\
[\![e\ \P\ \vec{t}]\!] &= \textbf{let}\ x = e\ \textbf{in}\ [\![\vec{t}]\!], x \notin \mathrm{fn}([\![\vec{t}]\!]) \\
[\![\textbf{if}\ e : \P\ \vec{t_1}\ \textbf{else}\ : \P\ \vec{t_2}]\!] &= \textbf{if}\ e\ \textbf{then}\ [\![\vec{t_1}]\!]\ \textbf{else}\ [\![\vec{t_2}]\!] \\
[\![\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}]\!] &= \{[\![\vec{f}]\!]\} \\
[\![x = e\ \P]\!] &= e \\
[\![e]\!] &= e \\
[\![\textbf{def}\ f(\vec{x}) : \P\ \vec{t}]\!] &= f(\vec{x}) = [\![\vec{t}]\!] \\
[\![\epsilon]\!] &= \{\}
\end{aligned}$$

Figure 7: Abstraction rules.

until the end of the current block, and that a block is defined by the indentation level of lines. The let construct, $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$, 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.

Figure 6 describes the abstract syntax for Callas. We retain just three syntactic categories: expressions e, modules M, and values v. On what concerns expressions, we add the **let** construct with the informal meaning explained above. Modules M are a collection of functions as before, but the new syntax allows for explicitly treating modules as values, and the module construct $\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}$ is expressed as a **let** for binding the module (the collection of functions) with variable x in a given scope, and as the name of the module itself. We also add a conditional $\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$ to expressions that evaluates condition e_2 when expression e_1 is true, and evaluates expression e_3 otherwise.

In Figure 7 we formalise the translations rules from concrete to abstract 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 $\vec{t_3}$). Notice that variable x plays no role in the continuation expression. A module $\textbf{module}\ x\ \textbf{of}\ T : \P\ \vec{f}$ 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

```
let s = {sample(self) = let curTemp = extern getTemp() in
                        send log(curTemp)} in
let mem = load in
let newMem = mem || s in
let skip = store newMem in
sample() every 60*10 expire 60*60*24*7
```

Figure 8: The abstract syntax of the sampling node.

$$
\begin{array}{lll@{\qquad}rcll}
S & ::= & \textit{Sensors} & R & ::= & e_1 :: \cdots :: e_n & \text{run-queue} \\
 & \mathbf{0} & \text{empty network} & T & ::= & \{(l_i(\vec{v_i}), v_i, v_i, v_i)\}_{i \in I} & \text{timed calls} \\
 & \mid S \,|\, S & \text{composition} & m & ::= & \langle l(\vec{v}) \rangle & \text{messages} \\
 & \mid [e, R \triangleright M, T]_t^{I,O} & \text{sensor} & I, O & ::= & m_1 :: \cdots :: m_n & \text{message queues}
\end{array}
$$

Figure 9: The syntax of Callas run-time environment.

$x = e$ ¶ at the head of a sequence of terms $\vec{t}$ is represented as a binding of expression e to x in the continuation $[\![\vec{t}]\!]$. Notice 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 Figure 1 we obtain the abstract syntax in Figure 8.

The run-time environment for Callas is presented in Figure 9 and focuses on the sensor components and on the network. Sensor networks S are concurrent compositions of sensors devices, represented as $[e, R \triangleright M, T]_t^{I,O}$, and of the empty network, denoted by $\mathbf{0}$. Each device is composed by an expression e being evaluated, 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), and the current time t. Messages are passivated function calls denoted as $\langle l(\vec{v}) \rangle$ and are the moving entities in the network.

2.3 Semantics

The meaning of programs in Callas is defined using an operational semantics, in particular a reduction system (Figure 11) defined with the help of a structural congruence relation (Figure 10).

Structural congruence identifies programs that are considered syntactically equivalent even when its textual representation is different. For instance, the equivalence $S_1 \,|\, S_2 \equiv S_2 \,|\, S_1$ means that sensor network programs obtained by permuting sensor representations are identical (in the sense that these programs possess the same computational meaning). The congruence rules are given in Figure 10 and the only non-standard rule is $[e, R \triangleright M, T]_t^{I,O} \equiv [e, R \triangleright M, T]_t^{I,O}\{\mathbf{0}\}$, which provides a conceptual *membrane* for the sensor. This *membrane* is present while the sensor is in communication and prevents sensors from receiving duplicate copies of a message during transmission.

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \qquad S \mid \mathbf{0} \equiv S, \qquad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3 \qquad \text{(S-monoid-Sensor)}$$

$$[e, R \triangleright M, T]_t^{I,O} \equiv [e, R \triangleright M, T]_t^{I,O}\{\mathbf{0}\} \qquad \text{(S-init-Send)}$$

Figure 10: Structural congruence for sensors.

The reduction relation is inductively defined by the rules in Figures 11 and 12. Since expressions evaluate to values, 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 evaluation 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 checked against the current clock time 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'. 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 non-blocking (Rule R-receiveE) and the program progresses even when there are no incoming messages. The condition expression **if** e_1 **then** e_2 **else** e_3 is standard. With Rule R-ifE (omitted) we let reduction occur in the condition, eventually resulting into a boolean value. Rule R-ifT (omitted) governs the case when the condition is True, in which case the conditional reduces into expression e_1 of the **then**-branch. Rule R-ifF (omitted) asserts the case when the condition is False, where the expression evolves into the **else**-branch expression e_2. Eventually expression e_1 evaluates to a value v (Rule R-letV) that replaces the free occurrences of variable x in e. When a programs evaluates to a value, it is discarded if there is another pending program in the run-queue, which in turn 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 copies the functions of both M and M' into a new module, giving preference to the functions of module M' in case of name clashes (*i.e.,* the same function name appearing on both modules). 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.

Rule R-timer programs a timer for a call to a function installed in the sensor device (whose code is in M). When predicate *noEvent* evaluates to false, rule R-interrupt comes into action, placing 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 (Figure 12) is orthogonal to that for in-sensor processing. Rules R-network and R-congr are straightforward. The former allows reduction

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

$$\frac{\text{noEvent}(T,t)}{[\langle \textsf{\textbf{load}}\ \rangle, R \triangleright M, T]_t^{I,O} \to [\langle M\rangle, R \triangleright M, T]_{t'}^{I,O}} \qquad \frac{\text{noEvent}(T,t)}{[\langle \textsf{\textbf{store}}\ \ v\rangle, R \triangleright M, T]_t^{I,O} \to [\langle \{\}\rangle, R \triangleright v, T]_{t'}^{I,O}}$$
$$(\textsc{R-load},\textsc{R-store})$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textsf{\textbf{send}}\ \ l(\vec{v})\rangle, R \triangleright M, T]_t^{I,O} \to [\langle \{\}\rangle, R \triangleright M, T]_{t'}^{I,O::\langle l(\vec{v})\rangle}} \qquad (\textsc{R-send})$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textsf{\textbf{receive}}\ \rangle, R \triangleright M, T]_t^{\langle l(\vec{v})\rangle::I,O} \to [\langle \{\}\rangle, R :: \textsf{\textbf{let}}\ \ x = \textsf{\textbf{load}}\ \ \textsf{\textbf{in}}\ \ x.l(\vec{v}) \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-receiveM})$$

$$\frac{\text{noEvent}(T,t)}{[\langle \textsf{\textbf{receive}}\ \rangle, R \triangleright M, T]_t^{\varepsilon,O} \to [\langle \{\}\rangle, R \triangleright M, T]_{t'}^{\varepsilon,O}} \qquad (\textsc{R-receiveE})$$

$$\frac{[e_1, R \triangleright M, T]_t^{I,O} \to [e_1', R' \triangleright M', T']_{t'}^{I',O'} \qquad \text{noEvent}(T,t)}{[\langle e_1\rangle, R \triangleright M, T]_t^{I,O} \to [\langle e_1'\rangle, R' \triangleright M', T']_{t'}^{I',O'}} \qquad (\textsc{R-let}E)$$

$$\frac{\text{noEvent}(T,t) \qquad x \neq y}{[\textsf{\textbf{let}}\ \ x = v\ \ \textsf{\textbf{in}}\ \ e, R \triangleright M, T]_t^{I,O} \to [\textsf{\textbf{let}}\ \ y = e[v/x]\ \ \textsf{\textbf{in}}\ \ y, R \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-let}V)$$

$$\frac{\text{noEvent}(T,t)}{[\textsf{\textbf{let}}\ \ x = v\ \ \textsf{\textbf{in}}\ \ x, e :: R \triangleright M, T]_t^{I,O} \to [e, R \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-next})$$

$$\frac{\text{noEvent}(T,t)}{[\textsf{\textbf{let}}\ \ x = v\ \ \textsf{\textbf{in}}\ \ x, \varepsilon \triangleright M, T]_t^{I,O} \to [\textsf{\textbf{let}}\ \ x = v\ \ \textsf{\textbf{in}}\ \ x, \varepsilon \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-idle})$$

$$\frac{\text{noEvent}(T,t) \qquad M''' = M' + M''}{[\langle M' \mid\mid M''\rangle, R \triangleright M, T]_t^{I,O} \to [\langle M'''\rangle, R \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-update})$$

$$\frac{M'(l) = l(\textsf{\textbf{self}}\ \ \vec{x})e' \quad \text{noEvent}(T,t)}{[\langle M'.l(\vec{v})\rangle, R \triangleright M, T]_t^{I,O} \to [\langle e'[M'\ \vec{v}/\textsf{\textbf{self}}\ \ \vec{x}]\rangle, R \triangleright M, T]_{t'}^{I,O}} \qquad (\textsc{R-call})$$

$$\frac{T' = T \uplus (l(\vec{v}), v, t+v, t+v') \qquad \text{noEvent}(T,t)}{[\langle l(\vec{v})\ \textsf{\textbf{every}}\ \ v\ \ \textsf{\textbf{expire}}\ \ v'\rangle, R \triangleright M, T]_t^{I,O} \to [\langle \{\}\rangle, R \triangleright M, T']_{t'}^{I,O}} \qquad (\textsc{R-timer})$$

$$\frac{t_1 \leq t_2 \qquad T' = T \uplus (l(\vec{v}), v, t_1+v, t_0)}{[e, R \triangleright M, T \uplus (l(\vec{v}), v, t_1, t_0)]_{t_2}^{I,O} \to [e, \textsf{\textbf{let}}\ \ x = \textsf{\textbf{load}}\ \ \textsf{\textbf{in}}\ \ x.l(\vec{v}) :: R \triangleright M, T']_{t_2}^{I,O}} \qquad (\textsc{R-interrupt})$$

$$\frac{t_1 > t_2 > t_0}{[e, R \triangleright M, T \uplus (l(\vec{v}), v, t_1, t_0)]_{t_2}^{I,O} \to [e, R \triangleright M, T]_{t_2}^{I,O}} \qquad (\textsc{R-expire})$$

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

Figure 11: Reduction semantics for sensors.

$$\frac{S \to S'}{S \mid S'' \to S' \mid S''} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \to S_3 \qquad S_3 \equiv S_4}{S_1 \to S_4} \qquad \text{(R-network, R-congr)}$$

$$\frac{(I'', O'') = \text{networkRoute}(m, I', O')}{[e, R \triangleright M, T]_t^{I,m::O}\{S\} \mid [e', R' \triangleright M', T']_{t'}^{I',O'} \to [e, R \triangleright M, T]_t^{I,m::O}\{S \mid [e', R' \triangleright M', T']_{t'}^{I'',O''}\}} \qquad \text{(R-broadcast)}$$

$$[e, R \triangleright M, T]_t^{I,m::O}\{S\} \to [e, R \triangleright M, T]_t^{I,O} \mid S \qquad \text{(R-release)}$$

Figure 12: Reduction semantics for sensor networks.

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 neighbourhood 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 predicate *networkRoute* that we leave unspecified. We omit modelling whether a given device is within communication range of the current broadcasting device. Predicate *networkRoute* implements the routing protocol used to propagate messages in the network, *e.g.*, the mesh mode in the Sun SPOT framework [14]. A transmission starts with the application of the structural congruence rule S-init-Send (Figure 10), continues with multiple applications of R-broadcast, and terminates with an application of R-release.

2.4 Type Safety

The static semantics for the Callas programming language is given in the form of a type-system [11] that we omit in the current paper. With this type-system and the reduction semantics from Figures 10, 11, and 12 we proved *subject-reduction*, *i.e.,* that types are invariant under reduction. We also proved the *type-safety* of the language, meaning that well-typed programs do not produce a class of run-time errors, namely that (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) updating a module preserves the signatures of the functions it contains.

Language type-safety is of utmost importance for WSN, since it allows the premature (static) detection of would-be run-time protocol errors, thus minimising the amount of debugging required for an application once it is deployed.

3 THE VIRTUAL MACHINE AND THE TRANSLATION

The Callas virtual machine (CVM) is a stack-based machine that serves as the run-time system for Callas. Instructions either push or pop values onto a stack to perform actions on the machine. CVM uses an incoming queue and an outgoing queue of passivated calls to interact with the lower layers of the network protocol stack. The machine state (see Figure 13) is divided into:

- an internal clock $\mathcal{Q}$ that uses arbitrary time units t;

machine state		$\mathcal{Q} \times \mathcal{M} \times \mathcal{T} \times \mathcal{C} \times \mathcal{R} \times \mathcal{I} \times \mathcal{O}$
time	$t \in$	$\mathcal{Q}$
timers	$\mathcal{T} \in$	$\textsc{SetOf}(\mathcal{S} \times \mathcal{Q} \times \mathcal{Q} \times \mathcal{Q})$
call-stack	$\mathcal{C} \in$	$\textsc{StackOf}(\mathsf{Int} \times \mathcal{E} \times \mathcal{S} \times \mathcal{B} \times \mathcal{U})$
passivated calls	$\mathcal{R}, \mathcal{I}, \mathcal{O} \in$	$\textsc{QueueOf}(l(\vec{v}))$
operands stack	$\mathcal{S} \in$	$\textsc{StackOf}(v)$

Figure 13: The syntactic categories of the virtual machine.

value	$v \in$	$\mathsf{Bool} \cup \mathsf{Int} \cup \mathsf{Float} \cup \mathsf{String} \cup \mathcal{M}$
instruction	$c \in$	$\{\texttt{update}, \texttt{extern}, \texttt{call}, \texttt{timer}, \texttt{return}, \texttt{jmp}\ n, \texttt{ift}\ n,$ $\texttt{receive}, \texttt{send}, \texttt{loadb}, \texttt{storeb}, \texttt{loadm}\ k, \texttt{loadc}\ k,$ $\texttt{load}\ k, \texttt{store}\ k, \texttt{binop}, \texttt{unop}\}$
program	$\mathcal{P} \in$	$\textsc{ArrayOf}(\mathcal{D})$
module declaration	$\mathcal{D} \in$	$\textsc{MapOf}(\mathsf{String} \mapsto \mathcal{F} \times \mathsf{Int})$
function declaration	$\mathcal{F} \in$	$\mathsf{Int} \times \mathsf{Int} \times \mathcal{B} \times \mathcal{U}$
module	$\mathcal{M} \in$	$\textsc{MapOf}(\mathsf{String} \mapsto \mathcal{F} \times \vec{v})$
values	$\mathcal{E}, \mathcal{U} \in$	$\textsc{ArrayOf}(v)$
byte-code	$\mathcal{B} \in$	$\textsc{ArrayOf}(c)$

Figure 14: The byte-code format.

- a module $\mathcal{M}$ that represents the shared memory of the device. The code of its functions can be altered during the execution of the device, but functions cannot be added or removed;

- a set of programmed timed function calls $\mathcal{T}$, each of which is divided into an operands stack, the period of the call, the time of the next call, and the expiration time;

- a call-stack $\mathcal{C}$ whose components, called call-frames, are divided into a program counter, an environment frame $\mathcal{E}$, an operands stack $\mathcal{S}$, a byte-code $\mathcal{B}$, and a constants array $\mathcal{U}$;

- a queue of pending calls $\mathcal{R}$.

- an incoming/outgoing queue $\mathcal{I}/\mathcal{O}$ of passivated calls.

The items in arrays, in stacks, and in queues of a syntactic category α are written $\langle \alpha_0, \ldots, \alpha_n \rangle$, $\alpha_0 : \cdots : \alpha_n$, and $\alpha_0 :: \cdots :: \alpha_n$, respectively. Empty arrays, stacks, and queues are denoted by ϵ.

CVM executes a low-level language (byte-code) that defines an instruction set (Figure 14). Instructions optionally accept one integer argument that is either one-byte long, denoted b, or four-bytes long, denoted n. A program $\mathcal{P}$ consists of an array of modules $\langle \vec{M} \rangle$. Each

$$
\begin{aligned}
&\llbracket e \rrbracket \stackrel{\text{def}}{=} \langle \vec{\mathcal{D}} \rangle \\
&\qquad \text{where } \vec{\mathcal{D}} = \mathtt{D}\llbracket M_1 \rrbracket(M_0 \dots M_n) \dots \mathtt{D}\llbracket M_n \rrbracket(M_0 \dots M_n) \\
&\qquad \text{and } M_0 \dots M_n = \text{modules}(\{\mathsf{run}(\,\textbf{self}\,) = e\}) +\!\!+ \{\}
\end{aligned}
$$

Figure 15: The translation of Callas programs.

module $\mathcal{M}$ in the array is a map from strings (the function names) onto tuples composed of an environment array $\mathcal{E}$, a byte-code array $\mathcal{B}$, and a constants array $\mathcal{U}$. The environment (or environment frame) is an array where the values of the local variables for a function are stored. The byte-code component contains the sequence of instructions $\vec{c}$ and ends with instruction `return`. There are instructions for manipulating modules, making calls, moving data, network I/O, control-flow, and basic arithmetic and logic operations. The constants array holds constants from the source program addressed by integer indexes, allowing for a simple and compact instruction set with few addressing modes. We present relevant instructions whilst describing the translation function.

3.1 The translation function

A translation function maps each term of the source language into zero or more terms of the target language. In Figure 15 we define the translation of expression e, denoted by $\llbracket e \rrbracket$, into a CVM program $\mathcal{P}$. For translating expression e into a program we identify modules $\vec{M}$ present in e and then translate each module M_i into a module declaration $\mathcal{D}_i$ via function $\mathtt{D}\llbracket M_i \rrbracket(\vec{M})$. The first declaration $\mathcal{D}_0$ consists of function run, which CVM executes upon starting. The byte-code of function run results from the translation of expression e described above. We define operator $+\!\!+$ for concatenating two sequences and filtering out repeated members. Function modules finds all modules in an expression tree, yielding a sequence. We concatenate the sequence of found modules with the empty module, because some expressions return an empty module, (*e.g.*, $\mathtt{E}\llbracket \textbf{receive} \rrbracket(\vec{x}, \vec{M}, \vec{v})$ in Figure 18).

As an example, we apply the translation function to the running example of the node broadcasting the sensed temperature listed in Figure 8. Let e be such program. We apply function modules to module $\{\mathsf{run}(\,\textbf{self}\,) \ = e\}$ and obtain a sequence of modules that consists of the argument of modules function and the only module found in e

$$\{\mathsf{run}(\,\textbf{self}\,) \ = e\}, \{\mathsf{sample}(\textbf{self}) = \textbf{let}\ \ \mathsf{currTemp} = \textbf{extern}\ \mathsf{getTemp()}\ \textbf{in}\ \textbf{send}\ \mathsf{log(curTemp)}\}$$

Let $\vec{M}$ be the concatenation of the modules above with the empty module $\{\}$. The generated program consists of an array holding three module declarations

$$
\begin{aligned}
&\langle \mathtt{D}\llbracket\{\mathsf{run}(\,\textbf{self}\,) \ = e\}\rrbracket(\vec{M}), \\
&\ \mathtt{D}\llbracket\{\mathsf{sample}(\textbf{self}) = \textbf{let}\ \ \mathsf{currTemp} = \textbf{extern}\ \mathsf{getTemp()}\ \textbf{in}\ \textbf{send}\ \mathsf{log(curTemp)}\}\rrbracket(\vec{M}), \\
&\ \mathtt{D}\llbracket\{\}\rrbracket(\vec{M})\rangle
\end{aligned}
$$

Function $\mathtt{D}\llbracket\cdot\rrbracket(\vec{M})$, defined in Figure 16, translates a module M into a module declaration $\mathcal{D}$. The translation function $\mathtt{F}\llbracket\cdot\rrbracket(\vec{M})$ yields a function declaration that is divided into a triple: an environment frame, whose size is computed by counting all variables present

$$
\begin{aligned}
\mathtt{D}[\![\{l_i(\vec{x}_i) = e_i\}_{i\in I}]\!](\vec{M}) &\overset{\text{def}}{=} \{l_i \mapsto \mathtt{F}[\![l_i(\vec{x}_i) = e_i]\!](\vec{M})\}_{i\in I} \\
\mathtt{F}[\![l(\vec{x}) = e]\!](\vec{M}) &\overset{\text{def}}{=} (|\vec{x}|, |\{\vec{y}\} \cup \mathrm{bn}(l(\vec{x}) = e)|, \langle\vec{c}\rangle, \langle\vec{v}\rangle), |\vec{y}| \\
&\qquad \text{where } \vec{y} = \mathrm{sort}(\mathrm{fn}(l(\vec{x}) = e)) \\
&\qquad \text{and } \vec{c} = \mathtt{E}[\![e]\!](\vec{x} +\!\!+\, \vec{y}, \vec{M}, \vec{v}), \mathtt{return} \\
&\qquad \text{and } \vec{v} = \mathrm{consts}(e)
\end{aligned}
$$

Figure 16: Translation of modules.

in e; the byte-code of the function; and all constants found in e computed by consts function (omitted). We append instruction `return` to the result of the translation function body. Function sort (not shown) takes a set of names and creates a sequence of names ordered lexicographically, to enable a predictable ordering of variables in the environment.

Continuing with the translation of the sampling node, we evaluate function $\mathtt{D}[\![\cdot]\!](\vec{M})$ and obtain the following program

$$
\begin{aligned}
&\langle\{\mathsf{run} \mapsto \mathtt{F}[\![\mathsf{run}(\textbf{\textsf{self}}) \ = e]\!](\vec{M})\}, \\
&\ \{\mathsf{sample} \mapsto \mathtt{F}[\![\mathsf{sample}(\textbf{\textsf{self}}) = \textbf{\textsf{let}}\ \ \mathsf{currTemp} = \textbf{\textsf{extern}}\ \mathsf{getTemp()}\ \textbf{\textsf{in send}}\ \mathsf{log(curTemp)}]\!](\vec{M})\}, \\
&\ \{\}\rangle
\end{aligned}
$$

Next we dwell on the translation of function run, $\mathtt{F}[\![\mathsf{run}(\textbf{\textsf{self}}) \ = e]\!](\vec{M})$ that yields the pair

$$
\begin{aligned}
&(1, 5, \langle \mathtt{E}[\![e]\!](\textbf{\textsf{self}}, \vec{M}, \vec{v}), \mathtt{return}\rangle, \langle\vec{v}\rangle), 0 \\
&\text{where } \vec{v} = \text{"}\mathsf{sample}\text{"}, \text{"}\mathsf{getTemp}\text{"}, \text{"}\mathsf{log}\text{"}, 60, 10, 24, 7
\end{aligned}
$$

The former member declares one parameter, a frame of five slots, a byte-code we discuss below, the sequence of constants $\vec{v}$ found in e (obtained by function consts). The latter specifies that there are no free variables.

Function $\mathtt{E}[\![\cdot]\!](\vec{x}, \vec{M}, \vec{v})$ generates byte-code from expressions. The parameters for this translation function are: the sequence $\vec{x}$ used for getting the index of a variable in the environment frame; the sequence of modules $\vec{M}$ used for obtaining the index of a module in the program; and the sequence of constants $\vec{v}$ used to obtain the index of a constant in the symbol array. The code generated for any expression leaves a value on top of the operands stack. The translation of a basic value, Figure 17, simply recurs to the position of the basic value in sequence $\vec{v}$ to obtain which index to utilise. For variables the position of the variable in the know variables $\vec{x}$ (calculated at compile-time) is the argument of the `load` instruction. When translating a module $\{l_i(\vec{x}_i) = e_i\}_{i\in I_j}$, the j-th module in the program, we push the closure of each function before loading the module (with `loadm`)—again, the position is given by the module's position in the sequence of known modules $\vec{M}$. To produce the code for a closure, we use function $\mathtt{A}[\![\cdot]\!](\vec{x}, \vec{M}, \vec{v})$ that translates each free variable as well as the name of the function.

Figure 18 describes the translation of expressions other than values; a rather straightforward and compact translation scheme. We comment just a few cases. The translation of a call $v.l(\vec{v})$ is the composition of translating its arguments augmented with module v, then its function name, and then instruction `call`. The translation of the arguments and of the

$$
\begin{aligned}
\mathtt{E}\llbracket v \rrbracket(\vec{x}, \vec{M}, v_0 \ldots v_i \ldots v_n) &\overset{\text{def}}{=} \mathtt{loadc}\ i \qquad \text{if } v = v_i \\
\mathtt{E}\llbracket x \rrbracket(x_0 \ldots x_i \ldots x_n, \vec{M}, \vec{v}) &\overset{\text{def}}{=} \mathtt{load}\ i \qquad \text{if } x = x_i \\
\mathtt{E}\llbracket \{l_i(\vec{x}_i) = e_i\}_{i \in I_j} \rrbracket(\vec{x}, \vec{M}, \vec{v}) &\overset{\text{def}}{=} \mathtt{A}\llbracket l_0(\mathrm{sort}(\mathrm{fn}((\vec{x}_0)e_0))) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \\
&\qquad \ldots \\
&\qquad \mathtt{A}\llbracket l_n(\mathrm{sort}(\mathrm{fn}((\vec{x}_n)e_n))) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \\
&\qquad \mathtt{loadm}\ j \\
&\qquad \text{where } \vec{M} \text{ is } M_0 \cdots \underbrace{\{l_i(\vec{x}_i) = e_i\}_{i \in I_j}}_{M_j} \cdots M_m \\
\mathtt{A}\llbracket l(v_0 \ldots v_k) \rrbracket(\vec{x}, \vec{M}, \vec{v}) &\overset{\text{def}}{=} \mathtt{E}\llbracket v_k \rrbracket(\vec{x}, \vec{M}, \vec{v}), \\
&\qquad \ldots \\
&\qquad \mathtt{E}\llbracket v_0 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \\
&\qquad \mathtt{E}\llbracket l \rrbracket(\vec{x}, \vec{M}, \vec{v})
\end{aligned}
$$

Figure 17: Translation of values and of sequences of values.

function name involves translating each argument in the right-to-left order (the *cdecl* calling convention), and then translating the function name. The case for **extern** $l(\vec{v})$ is similar to a function invocation, except that, because this is a direct system call, we do not place the target module in the first argument. The timed call expression $l(\vec{v})$ **every** e **expire** e' has a slightly more contrived translation. First, we translate the expiration time, the period of the call, the arguments, and the name of the function. Thereafter we issue `timer` instruction to program the timer. Lastly, we translate the empty module, for placing the result of the operation onto the operands stack, since the operation always returns the empty module $\{\}$.

To give a flavour of how the translation algorithm develops, we translate the arithmetic expression $24 * 7$, an instance of a binary operation. Henceforth, for the remaining examples let

$$\vec{v} = \text{"}\mathit{sample}\text{"}, \text{"}\mathsf{getTemp}\text{"}, \text{"}\mathsf{log}\text{"}, 60, 10, 24, 7$$

The translation is valid for any sequence of names $\vec{x}$, as well as for any sequence of modules $\vec{M}$.

$$\mathtt{E}\llbracket \mathsf{24{*}7} \rrbracket(\vec{x}, \vec{M}, \vec{v}) = \mathtt{loadc}\ 6, \mathtt{loadc}\ 5, \mathtt{mul}$$

The outcome is pushing constant 7 and then pushing constant 24 to the operands stack; then, by issuing a `mul`, the machine takes both numbers from the top of the stack, performs the multiplication, and pushes number 168 onto the stack.

A composite arithmetic expression like $\mathsf{60{*}60{*}24{*}7}$ is more complex, but standard for stack-based machines. We annotate each instruction with its respective source operand or operator to ease the comprehension.

$$\mathtt{E}\llbracket \mathsf{60{*}60{*}24{*}7} \rrbracket(\vec{x}, \vec{M}, \vec{v}) = \underbrace{\mathtt{loadc}\ 6}_{7}, \underbrace{\mathtt{loadc}\ 5}_{24}, \underbrace{\mathtt{mul}}_{*}, \underbrace{\mathtt{loadc}\ 3}_{60}, \underbrace{\mathtt{mul}}_{*}, \underbrace{\mathtt{loadc}\ 3}_{60}, \underbrace{\mathtt{mul}}_{*}$$

$$\mathtt{E}\llbracket \textsf{\textbf{load}} \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \ \mathtt{loadb}$$

$$\mathtt{E}\llbracket \textsf{\textbf{store}}\ \ v \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{E}\llbracket v \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{storeb}, \mathtt{E}\llbracket \{\} \rrbracket(\vec{x}, \vec{M}, \vec{v})$$

$$\mathtt{E}\llbracket v_1 \ || \ v_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{E}\llbracket v_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{E}\llbracket v_1 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{update}$$

$$\mathtt{E}\llbracket v_1 \ \ \mathrm{binop} \ \ v_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{E}\llbracket v_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{E}\llbracket v_1 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{binop}$$

$$\mathtt{E}\llbracket \mathrm{unop} \ \ v \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{E}\llbracket v \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{unop}$$

$$\mathtt{E}\llbracket v.l(\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{A}\llbracket l(v\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{call}$$

$$\mathtt{E}\llbracket \textsf{\textbf{extern}} \ \ l(\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{A}\llbracket l(\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{extern}$$

$$\begin{aligned}&\mathtt{E}\llbracket l(\vec{v}) \ \textsf{\textbf{every}} \ \ e_1 \ \textsf{\textbf{expire}} \ \ e_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \\ &\quad \mathtt{A}\llbracket l(\vec{v}e_1e_2) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{timer}, \mathtt{E}\llbracket \{\} \rrbracket(\vec{x}, \vec{M}, \vec{v})\end{aligned}$$

$$\mathtt{E}\llbracket \textsf{\textbf{send}} \ \ l(\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{A}\llbracket l(\vec{v}) \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{send}, \mathtt{E}\llbracket \{\} \rrbracket(\vec{x}, \vec{M}, \vec{v})$$

$$\mathtt{E}\llbracket \textsf{\textbf{receive}} \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \mathtt{receive}, \mathtt{E}\llbracket \{\} \rrbracket(\vec{x}, \vec{M}, \vec{v})$$

$$\begin{aligned}&\mathtt{E}\llbracket \textsf{\textbf{let}} \ \ x = e_1 \ \textsf{\textbf{in}} \ \ e_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \\ &\quad \mathtt{E}\llbracket e_1 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{store}\ i, \mathtt{E}\llbracket e_2 \rrbracket(y_1 \ldots y_i \ldots y_n, \vec{M}, \vec{v}) \quad \text{if } x = y_i \\ &\quad \text{where } y_1 \ldots y_i \ldots y_n = \vec{x} +\!\!+\, x\end{aligned}$$

$$\begin{aligned}&\mathtt{E}\llbracket \textsf{\textbf{if}} \ \ e_1 \ \textsf{\textbf{then}} \ \ e_2 \ \textsf{\textbf{else}} \ \ e_3 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \overset{\text{def}}{=} \\ &\quad \mathtt{E}\llbracket e_1 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \mathtt{ift}\ i_3, \mathtt{E}\llbracket e_3 \rrbracket(\vec{x}, \vec{M}, \vec{v}), \\ &\quad \mathtt{jmp}\ i_2, \mathtt{E}\llbracket e_2 \rrbracket(\vec{x}, \vec{M}, \vec{v}) \\ &\quad \text{where } i_2 = |\mathtt{E}\llbracket e_2 \rrbracket(\vec{x}, \vec{M}, \vec{v})| \\ &\qquad \text{and } i_3 = |\mathtt{E}\llbracket e_3 \rrbracket(\vec{x}, \vec{M}, \vec{v})| + 1\end{aligned}$$

Figure 18: Translation of expressions (except values).

We create a module by specifying the value of each free variable it holds and then issuing `loadm`. Translating the empty module, in the third position of $\vec{M}$, consists of only one instruction.

$$\mathtt{E}[\![\{\}]\!](\vec{x},\vec{M},\vec{v}) = \mathtt{loadm}\ 2$$

Translating the timed call $\mathsf{sample()}\ \textsf{\textbf{every}}\ \mathsf{60{*}10}\ \textsf{\textbf{expire}}\ \mathsf{60{*}60{*}24{*}7}$ pushes expression $60*60*24*7$, expression $60*10$, and the function's name (plus zero arguments); then issues a `timer`, and finishes by pushing the empty module onto the stack.

$$\begin{aligned}
&\mathtt{E}[\![\mathsf{sample()}\ \textsf{\textbf{every}}\ \mathsf{60{*}10}\ \textsf{\textbf{expire}}\ \mathsf{60{*}60{*}24{*}7}]\!](\vec{x},\vec{M},\vec{v}) = \\
&\qquad \underbrace{\mathtt{loadc}\ 6, \mathtt{loadc}\ 5, \mathtt{mul}, \mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{loadc}\ 3, \mathtt{mul}}_{\mathtt{E}[\![60*60*24*7]\!](\ldots)}, \\
&\qquad \underbrace{\mathtt{loadc}\ 4, \mathtt{loadc}\ 3, \mathtt{mul}}_{\mathtt{E}[\![60*10]\!](\ldots)}, \underbrace{\mathtt{loadc}\ 0}_{\mathsf{sample}}, \mathtt{timer}, \underbrace{\mathtt{loadm}\ 2}_{\mathtt{E}[\![\{\}]\!](\ldots)}
\end{aligned}$$

We translate expression $\textsf{\textbf{store}}\ \mathsf{newMem}$ that stores a module from variable newMem in the memory of the device. The generated code is straightforward; we load the module in slot 3 (the fourth know variable), then issue a `storeb` that pops the module on top of the operands stack and uses it as the installed code of the device, and then push an empty module onto the operands stack (the result of the expression).

$$\mathtt{E}[\![\textsf{\textbf{store}}\ \mathsf{newMem}]\!]((\textsf{\textbf{self}}, \mathsf{s}, \mathsf{mem}, \mathsf{newMem}), \vec{M}, \vec{v}) = \mathtt{load}\ 3, \mathtt{storeb}, \mathtt{loadm}\ 2$$

Let expression e_1 be

$$\textsf{\textbf{let}}\ \mathsf{skip} = \textsf{\textbf{store}}\ \mathsf{newMem}\ \textsf{\textbf{in}}\ \mathsf{sample()}\ \textsf{\textbf{every}}\ \mathsf{60{*}10}\ \textsf{\textbf{expire}}\ \mathsf{60{*}60{*}24{*}7}$$

Translating the composite expression e_1 amounts to translate $\textsf{\textbf{store}}\ \mathsf{newMem}$, then issue instruction `store` to copy the value on top of the operands stack into the slot with index 4 of the environment (given by the position of variable skip in the sequence of known variables), and then translating the timed call.

$$\begin{aligned}
\mathtt{E}[\![e_1]\!]((\textsf{\textbf{self}}, \mathsf{s}, \mathsf{mem}, \mathsf{newMem}), \vec{M}, \vec{v}) = {} & \underbrace{\mathtt{load}\ 3, \mathtt{storeb}, \mathtt{loadm}\ 2}_{\mathtt{E}[\![\textsf{\textbf{store}}\ \mathsf{newMem}]\!](\ldots)}, \underbrace{\mathtt{store}\ 4}_{\mathsf{skip}}, \underbrace{\mathtt{loadc}\ 0}_{\mathsf{sample}}, \\
& \underbrace{\mathtt{loadc}\ 4, \mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{loadc}\ 6, \mathtt{loadc}\ 5, \mathtt{mul},}_{\mathtt{E}[\![\mathsf{sample()}\ \textsf{\textbf{every}}\ \mathsf{60{*}10}\ \textsf{\textbf{expire}}\ \mathsf{60{*}60{*}24{*}7}]\!](\ldots)} \\
& \underbrace{\mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{timer}, \mathtt{loadm}\ 2}_{\text{(continuation) } \mathtt{E}[\![\mathsf{sample()}\ \textsf{\textbf{every}}\ \mathsf{60{*}10}\ \textsf{\textbf{expire}}\ \mathsf{60{*}60{*}24{*}7}]\!](\ldots)} \\
= {} & \vec{c}_1
\end{aligned}$$

The translation of $\textsf{\textbf{let}}\ \mathsf{newMem} = \mathsf{mem}\ ||\ \mathsf{s}\ \textsf{\textbf{in}}\ e_1$ is similar to translating a binary arithmetic operation, thus we translate variables s and mem, next we issue instruction `update` that takes two modules from the top of the operands stack and merges them into a new module that is pushed on the operands stack. Afterwards we store the new module in slot with index 3, reserved for variable newMem, and translate expression e_1 whose result is sequence $\vec{c}_1$.

$$\mathtt{E}[\![\underbrace{\textsf{\textbf{let}}\ \mathsf{newMem} = \mathsf{mem}\ ||\ \mathsf{s}\ \textsf{\textbf{in}}\ e_1}_{e_2}]\!]((\textsf{\textbf{self}}, \mathsf{s}, \mathsf{mem}), \vec{M}, \vec{v}) = \mathtt{load}\ 1, \mathtt{load}\ 2, \mathtt{update}, \mathtt{store}\ 3, \vec{c}_1$$

The new module, created by instruction `update`, is composed of the functions in both operands (on the top of the operands stack). In case there exists a function in the first operand with the same name than a function in the second operand, the function in second operand is chosen to compose the new module.

Translating expression $\textbf{let}\ \mathsf{mem} = \textbf{load in}\ e_2$ that loads the memory of the device into variable mem and continues with expression e_2 is trivial, where expression $e_2 = \textbf{let}\ \mathsf{newMem} = \mathsf{mem}\ ||\ \mathsf{s}\ \textbf{in}\ e_1$.

$$\mathtt{E}[\![\textbf{let}\ \mathsf{mem} = \textbf{load in}\ e_2]\!](\textbf{self}\ \mathsf{s}, \vec{M}, \vec{v}) = \mathtt{loadb}, \mathtt{store}\ 2, \underbrace{\underbrace{\mathtt{load}\ 1, \mathtt{load}\ 2, \mathtt{update}, \mathtt{store}\ 3, \vec{c}_1}_{\mathtt{E}[\![e_2]\!](\dots)}}_{\vec{c}_2}$$

We translate expression $\textbf{load}$ into instruction `loadb` that pushes the installed code onto the operands stack.

Let module $M_2 = \{\mathsf{sample}(\textbf{self}) = \textbf{let}\ \mathsf{currTemp} = \textbf{extern}\ \mathsf{getTemp}()\ \textbf{in send}\ \mathsf{log}(\mathsf{curTemp})\}$. To generate code for creating this module, we push name sample along with the free variables of the function. In this case there are none, so we just push the function name prior to issuing instruction `loadm`.

$$\mathtt{E}[\![M_2]\!](\textbf{self}, \vec{M}, \vec{v}) = \mathtt{loadc}\ 0, \mathtt{loadm}\ 1$$

To translate expression $\textbf{let}\ \mathsf{s} = M_2\ \textbf{in}\ e_3$ that creates module M_2, saves it in variable s, and proceeds with expression e_3, where $e_3 = \textbf{let}\ \mathsf{mem} = \textbf{load in}\ e_2$, we translate module M_2 (as above) and proceed with the translation of e_3 (sequence $\vec{c}_2$).

$$\mathtt{E}[\![\textbf{let}\ \mathsf{s} = M_2\ \textbf{in}\ e_3]\!](\textbf{self}, \vec{M}, \vec{v}) = \mathtt{loadc}\ 0, \mathtt{loadm}\ 1, \mathtt{store}\ 1, \vec{c}_2$$

Going back to the example of the module declaration for module M_1, we have that

$$\begin{aligned}\mathtt{D}[\![M_1]\!](\vec{M}) = \{\mathsf{run} = (\langle 0\rangle, &\langle \mathtt{loadc}\ 0, \mathtt{loadm}\ 1, \mathtt{store}\ 1, \vec{c}_2, \mathtt{return}\rangle,\\ &\langle \text{"sample"}, \text{"getTemp"}, \text{"log"}, 60, 10, 24, 7\rangle)\}\end{aligned}$$

Thus, the translated program follows

$$\begin{aligned}\langle\{&\mathsf{run} = (1, 5, \langle \mathtt{loadc}\ 0, \mathtt{loadm}\ 1, \mathtt{store}\ 1, \vec{c}_2, \mathtt{return}\rangle,\\ &\quad\langle \text{"sample"}, \text{"getTemp"}, \text{"log"}, 60, 10, 24, 7\rangle), 0\},\\ \{&\mathsf{sample} = (1, 2, \langle \mathtt{loadc}\ 0, \mathtt{extern}, \mathtt{store}\ 1, \mathtt{load}\ 1, \mathtt{loadc}\ 1, \mathtt{send}, \mathtt{loadm}\ 2, \mathtt{return}\rangle\\ &\qquad\langle \text{"getTemp"}, \text{"log"}\rangle), 0\},\\ \{\}&\rangle\end{aligned}$$

We omit the detailed translation of module sample.

3.2 The operational semantics of CVM

The entry point of every program is the function that is executed when the machine starts, namely run function of the first module in the program. The function declaration for run must have one parameter, a frame of size $k+1$, a byte-code $\mathcal{B}$, and a constant array $\mathcal{U}$; run function of this module must not hold free variables.

$$\mathcal{P}[0](\mathsf{run}) = \underbrace{(1, k+1, \mathcal{B},\mathcal{U})}_{\mathcal{F}_{\mathsf{run}}}, 0$$

The initial state of a machine running $\mathcal{P}$ where $|\vec{0}| = k$ is

$$\langle 0, \{\}, \{\}, (0, \langle \{\mathsf{run} \mapsto \mathcal{F}_{\mathsf{run}}\}, \vec{0} \rangle, \epsilon, \mathcal{B},\mathcal{U}), \epsilon \rangle^{\epsilon}_{\epsilon}$$

The first member is the machine's current time and starts at zero. The second member, the installed code, begins with no functions stored. The third member is a pool of timed events (initially empty). The fourth member is the call-stack that starts with a frame holding: a program counter pointing to the first instruction, an environment that consists of the first module in the program and zeroes, an empty operands stack, byte-code $\mathcal{B}$ of function run, and constant pool $\mathcal{U}$. The fifth member is the run-queue and commences with no pending calls. Finally, the incoming- and outgoing-queues initiate cleared.

The execution of the program proceeds through a series of state transitions, presented in Figures 19 and 20, that were designed to match the operational semantics given in Section 2. Rule R-EXTERN performs a synchronous invocation to an external function l via instruction `extern`, for an operating system call. The virtual machine removes function name l and arguments $\vec{v}$ from the top of the operands stack, respectively. Function *externArgsCount* maps a function name to the number of arguments the external function accepts. Function *callExtern* abstracts the execution by producing value v, placed at the top of the operand stack. Instructions `send` and `receive` handle the interaction of CVM with the network layer. Instruction `send` (Rule R-SEND) takes a function name l and a sequence of values $\vec{v}$ from the operands stack into output queue O. Instruction `receive` takes a passivated function call $l(\vec{v})$ from the nonempty input queue I into run-queue $\mathcal{R}$ (Rule R-RECEIVEM). Rule R-RECEIVEE asserts that instruction `receive` is non-blocking. Thus, if the input queue is empty, CVM just increments the program counter. Rule R-LOAD copies the k-th value of environment $\mathcal{E}$ to the top of the operands stack. Instruction `store` k copies value v on top of the operands stack into the k-th slot of the environment and then pops the top of the operands stack (Rule R-STORE). With Rule R-LOADC, we push the k-th value of constant array $\mathcal{U}$ into the operands stack.

Instruction `loadb` pushes the installed code $\mathcal{M}$ onto the operands stack (Rule R-LOADB). Instruction `storeb` takes module $\mathcal{M}_1$ from the top of the operands stack and uses it as the installed code of the device (Rule R-STOREB). Rule R-LOADM pushes a new module from the k-th declaration in program $\mathcal{P}$. In the operands stack, for each function l_i we expect the label of the function and closure $\vec{v}_i$; the length of the closure (k_i) is specified by entry $\mathcal{D}(l_i)$. The i-th function of the created module holds function declaration $\mathcal{F}_i$ (given by the module declaration) and closure $\vec{v}_i$. Instruction `update` takes modules $\mathcal{M}_1$ and $\mathcal{M}_2$ from the top of the operands stack and merges them into a new module $\mathcal{M}_3$ that is pushed on the operands stack (Rule R-UPDATE). Module $\mathcal{M}_3$ consists of the functions in $\mathcal{M}_1$ plus the functions in $\mathcal{M}_2$. In case there exists a function in $\mathcal{M}_1$ with the same name as a function in $\mathcal{M}_2$, the latter is chosen to compose module $\mathcal{M}_3$. We instruction `call` invokes functions in modules and expects a module $\mathcal{M}_1$, a string l, and a sequence of arguments $\vec{v}_1$ on top of the operands stack (Rule R-CALL). We use string l to select a function in module $\mathcal{M}_1$ and obtain the number of parameters to take from the operands stack, the size of the frame we are creating, the byte

$$\frac{\mathcal{B}[i] = \texttt{extern} \qquad k \leftarrow \mathit{externArgsCount}(l) \qquad |\vec{v}| = k \qquad v \leftarrow \mathit{callExtern}(l : \vec{v})}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, l : \vec{v} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{extern})}$$

$$\frac{\mathcal{B}[i] = \texttt{send} \qquad \mathcal{M}(l) = (|\vec{v}| + 1, \_, \_, \_), \_}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, l : \vec{v} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}::l(\vec{v})}} \quad \text{(R-\textsc{send})}$$

$$\frac{\mathcal{B}[i] = \texttt{receive}}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{l(\vec{v})::\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, R :: l(\vec{v})\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{receive}M)}$$

$$\frac{\mathcal{B}[i] = \texttt{receive}}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\epsilon}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\epsilon}_{\mathcal{O}}} \quad \text{(R-\textsc{receive}E)}$$

$$\frac{\mathcal{B}[i] = \texttt{load}\ k \qquad \mathcal{E}[k] = v}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{load})}$$

$$\frac{\mathcal{B}[i] = \texttt{store}\ k \qquad \mathcal{E}[k] \leftarrow v}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{store})}$$

$$\frac{\mathcal{B}[i] = \texttt{loadc}\ k \qquad \mathcal{U}[k] = v}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{load}C)}$$

$$\frac{\mathcal{B}[i] = \texttt{loadb}}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{M} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{load}B)}$$

$$\frac{\mathcal{B}[i] = \texttt{storeb}}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{M}_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}_1, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{store}B)}$$

$$\frac{\mathcal{B}[i] = \texttt{loadm}\ k \qquad \mathcal{P}[k] = \mathcal{D} \qquad |\mathcal{D}| = n \qquad \mathcal{D}(l_i) = \mathcal{F}_i, k_i \qquad |\vec{v}_i| = k_i}{\begin{array}{l} \langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, l_0 : \vec{v_0} : \cdots : l_n : \vec{v_n} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad\qquad \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \{l_i \mapsto \mathcal{F}_i, \vec{v}_i\}_{i \in 0 \ldots n} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \end{array}} \quad \text{(R-\textsc{load}M)}$$

$$\frac{\mathcal{B}[i] = \texttt{update} \qquad \mathcal{M}_1 + \mathcal{M}_2 = \mathcal{M}_3}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{M}_1 : \mathcal{M}_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i+1, \mathcal{E}, \mathcal{M}_3 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad \text{(R-\textsc{update})}$$

$$\frac{\mathcal{B}[i] = \texttt{call} \qquad \mathcal{M}_1(l) = (|\mathcal{M}_1 \vec{v}_1|, |\mathcal{M}_1 \vec{v}_1 \vec{v}_2 \vec{0}|, \mathcal{B}', \mathcal{U}'), \vec{v}_2 \qquad \mathcal{E}' = \langle \mathcal{M}_1 \vec{v}_1 \vec{v}_2 \vec{0} \rangle}{\begin{array}{l} \langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, l : \mathcal{M}_1 : \vec{v}_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad\qquad \langle t', \mathcal{M}, \mathcal{T}, (0, \mathcal{E}', \epsilon, \mathcal{B}', \mathcal{U}') : (i+1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \end{array}} \quad \text{(R-\textsc{call})}$$

Figure 19: Reduction rules for CVM: system, network, data, modules and calls.

$$\frac{\mathcal{B}[i] = \texttt{timer} \qquad \mathcal{M}(l) = (|\vec{v}| + 1, \_, \_, \_), \_}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, l : \vec{v} : t_1 : t_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad \langle t', \mathcal{M}, \mathcal{T} \uplus \{(l(\vec{v}), t_1, t + t_1, t + t_2)\}, (i + 1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}} \quad (\textsc{R-timer})$$

$$\frac{t_1 \leq t}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T} \uplus \{(l(\vec{v}), t_0, t_1, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad \langle t, \mathcal{M}, \mathcal{T} \uplus \{(l(\vec{v}), t_0, t_1 + t_0, t_2)\}, \mathcal{C}, l(\vec{v}) :: \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}} \quad (\textsc{R-interrupt})$$

$$\frac{t_1 > t > t_2}{\langle t, \mathcal{M}, \mathcal{T} \uplus \{(\mathcal{S}, \_, t_1, t_2)\}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t, \mathcal{M}, \mathcal{T}, \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-expire})$$

$$\frac{\mathcal{B}[i] = \texttt{return}}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : (i', \mathcal{E}', \mathcal{S}', \mathcal{B}', \mathcal{U}') : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad \langle t', \mathcal{M}, \mathcal{T}, (i', \mathcal{E}', v : \mathcal{S}', \mathcal{B}', \mathcal{U}') : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}} \quad (\textsc{R-return})$$

$$\frac{\mathcal{B}[i] = \texttt{return} \qquad \mathcal{B}' = \langle \texttt{loadb}, \texttt{loadc}\, 0, \texttt{call}, \texttt{return}\rangle}{\begin{array}{l}\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}), l(v_0, \ldots, v_n) :: R\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \\ \qquad \langle t', \mathcal{M}, \mathcal{T}, (0, \epsilon, v_0 : \cdots : v_n, \mathcal{B}', \langle l\rangle), \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}\end{array}} \quad (\textsc{R-next})$$

$$\frac{\mathcal{B}[i] = \texttt{return}}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}), \epsilon\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}), \epsilon\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-idle})$$

$$\frac{\mathcal{B}[i] = \texttt{jmp}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i + 1 + n, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-jump})$$

$$\frac{\mathcal{B}[i] = \texttt{ift}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathsf{True} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i + 1 + n, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-ifT})$$

$$\frac{\mathcal{B}[i] = \texttt{ift}\ n}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, \mathsf{False} : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i + 1, \mathcal{E}, \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-ifF})$$

$$\frac{\mathcal{B}[i] = \texttt{binop} \qquad v \leftarrow \mathrm{binop}(\texttt{binop}, v_1, v_2)}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v_1 : v_2 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i + 1, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-bin})$$

$$\frac{\mathcal{B}[i] = \texttt{unop} \qquad v \leftarrow \mathrm{unop}(\texttt{unop}, v_1)}{\langle t, \mathcal{M}, \mathcal{T}, (i, \mathcal{E}, v_1 : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}} \rightarrow_{\mathcal{P}} \langle t', \mathcal{M}, \mathcal{T}, (i + 1, \mathcal{E}, v : \mathcal{S}, \mathcal{B}, \mathcal{U}) : \mathcal{C}, \mathcal{R}\rangle^{\mathcal{I}}_{\mathcal{O}}} \quad (\textsc{R-un})$$

Figure 20: Reduction rules for CVM: timers, control flow, binary and unary operators.

code to execute, the constant pool, and the closure of this function. The new environment $\mathcal{E}'$ is composed by the function's arguments $\mathcal{M}_1\vec{v}_1$ (from the operands stack), the function's closure $\vec{v}_2$ (from function l), and some zeros (the amount is calculated using the frame size minus the arguments count and the closure length). The resulting call-frame is placed on top of the call-stack C.

We use instruction `timer` to program an event that periodically schedules a call to installed function l with arguments $\vec{v}$ (Rule R-timer). The periodicity t_1 and the duration of the timer t_2 are taken from the operands stack. Rule R-interrupt precises how CVM monitors the system clock, placing a function call in front of the run queue $\mathcal{R}$ every time a period has passed since the last trigger. The monitoring is done on a best effort basis, meaning that the calls may not be strictly periodic and there may be some drift over time. With Rule R-expire the machine discards expired timers.

CVM functions always return a value. The last instruction of every function is a `return` that expects an operands stack with a return value to pass to the calling frame. In case there is a call-frame below, *i.e.,* a function calling another function, the top frame is discarded and the return value is pushed onto the operands stack of the calling frame (Rule R-return). Otherwise, in case there is no call-frame below, the machine waits for the existence of a pending call in the run queue (Rule R-idle) and then (Rule R-next) constructs a new call-frame that consists of: a program counter pointing to the first instruction of the byte-code, an operands stack holding the arguments of the function, the byte-code necessary to invoke function l from the installed module, and a constant pool with the function's name.

Instructions `jmp` and `ift` are the unconditional and the conditional branching, respectively. Both use a relative address to obtain the next instruction to execute, rather than an absolute addresses. Rules R-bin and R-un concern binary and unary operations, respectively. In these rules the instruction itself is a parameter of the rule, meaning that, for example Rule R-bin, applies to zero or more instructions—like the addition operator, or the subtraction operator. These are the typical operations over primitive values (*e.g.*, arithmetic, or logic) commonly found on stack-based architectures.

To illustrate the workings of the virtual machine, take the program generated in the previous section:

$$
\begin{aligned}
\mathcal{P} =& \langle \{\mathsf{run} \mapsto \underbrace{(1, 5, \mathcal{B}, \mathcal{U})}_{\mathcal{F}_0}, 0\}, \{\mathsf{sample} \mapsto \underbrace{(1, 2, \mathcal{B}', \mathcal{U}')}_{\mathcal{F}_1}\}, \{\}\rangle \\
\mathcal{B} =& \langle \mathtt{loadc}\ 0, \mathtt{loadm}\ 1, \mathtt{store}\ 1, \mathtt{loadb}, \mathtt{store}\ 2, \mathtt{load}\ 1, \mathtt{load}\ 2, \mathtt{update}, \mathtt{store}\ 3, \\
& \quad \mathtt{load}\ 3, \mathtt{storeb}, \mathtt{loadm}\ 2, \mathtt{store}\ 4, \mathtt{loadc}\ 6, \mathtt{loadc}\ 5, \mathtt{mul}, \mathtt{loadc}\ 3, \mathtt{mul}, \\
& \quad \mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{loadc}\ 4, \mathtt{loadc}\ 3, \mathtt{mul}, \mathtt{loadc}\ 0, \mathtt{timer}, \mathtt{loadm}\ 2, \mathtt{return}\rangle \\
\mathcal{U} =& \langle \text{"}\mathsf{sample}\text{"}, \text{"}\mathsf{getTemp}\text{"}, \text{"}\mathsf{log}\text{"}, 60, 10, 24, 7\rangle \\
\mathcal{B}' =& \langle \mathtt{loadc}\ 0, \mathtt{extern}, \mathtt{store}\ 1, \mathtt{load}\ 1, \mathtt{loadc}\ 1, \mathtt{send}, \mathtt{loadm}\ 2, \mathtt{return}\rangle \\
\mathcal{U}' =& \langle \text{"}\mathsf{getTemp}\text{"}, \text{"}\mathsf{log}\text{"}\rangle
\end{aligned}
$$

and we execute, step-by-step, the byte-code for the function run, as generated in Section 3.1. The initial state CVM for the translated program is

$$\langle 0, \{\}, \{\}, (0, \langle \underbrace{\{\mathsf{run} \mapsto \mathcal{F}_0, \langle\rangle\}}_{\mathcal{M}_0}, 0, 0, 0, 0\rangle, \epsilon, \mathcal{B}, \mathcal{U}), \epsilon\rangle_{\epsilon}^{\epsilon}$$

The virtual machine executes the byte-code sequentially, at each step examining the instruction indexed by the program counter and performing the associated changes in the state. The following listing shows a trace of the execution. The first two executions steps create a module from the module declaration at slot 1 and stores the module in index 0 of the environment frame. The first reduction pushes the free variables of sample function; since there are none, we just push the function's name. The second instruction issues instruction $\mathtt{loadm}$, pushing $M_1 = \{\mathsf{sample} \mapsto \mathcal{F}_1, \langle\rangle\}$ onto the operands stack.

$$\begin{array}{l}
\langle 0, \{\}, \{\}, (0, \langle \mathcal{M}_0, 0, 0, 0, 0\rangle, \epsilon, \langle \mathtt{loadc}\ 0, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 1, \{\}, \{\}, (1, \langle \mathcal{M}_0, 0, 0, 0, 0\rangle, "\mathsf{sample}", \langle \ldots, \mathtt{loadm}\ 1, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 2, \{\}, \{\}, (2, \langle \mathcal{M}_0, 0, 0, 0, 0\rangle, \mathcal{M}_1, \langle \ldots, \mathtt{store}\ 1, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon}
\end{array}$$

The subsequent eight reduction steps update the memory of the sensor (an empty module) with the module $\mathcal{M}_1$ stored in slot 1. Note that $\{\} + \mathcal{M}_1 = \mathcal{M}_1$.

$$\begin{array}{l}
\langle 3, \{\}, \{\}, (3, \langle \mathcal{M}_0, \mathcal{M}_1, 0, 0, 0\rangle, \epsilon, \langle \ldots, \mathtt{loadb}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 4, \{\}, \{\}, (4, \langle \mathcal{M}_0, \mathcal{M}_1, 0, 0, 0\rangle, \{\}, \langle \ldots, \mathtt{store}\ 2, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 5, \{\}, \{\}, (5, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, 0, 0\rangle, \epsilon, \langle \ldots, \mathtt{load}\ 1, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 6, \{\}, \{\}, (6, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, 0, 0\rangle, \mathcal{M}_1, \langle \ldots, \mathtt{load}\ 2, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 7, \{\}, \{\}, (7, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, 0, 0\rangle, \{\} : \mathcal{M}_1, \langle \ldots, \mathtt{update}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 8, \{\}, \{\}, (8, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, 0, 0\rangle, \mathcal{M}_1, \langle \ldots, \mathtt{store}\ 3, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 9, \{\}, \{\}, (9, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, \mathcal{M}_1, 0\rangle, \epsilon, \langle \ldots, \mathtt{load}\ 3, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 10, \{\}, \{\}, (10, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, \mathcal{M}_1, 0\rangle, \mathcal{M}_1, \langle \ldots, \mathtt{storeb}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow
\end{array}$$

The next two instructions store an empty module in slot 4.

$$\begin{array}{l}
\langle 11, \mathcal{M}_1, \{\}, (11, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, \mathcal{M}_1, 0\rangle, \epsilon, \langle \ldots, \mathtt{loadm}\ 2, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 12, \mathcal{M}_1, \{\}, (12, \langle \mathcal{M}_0, \mathcal{M}_1, \{\}, \mathcal{M}_1, 0\rangle, \{\}, \langle \ldots, \mathtt{store}\ 4, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon}
\end{array}$$

The posterior execution is the translation of $\mathtt{E}[\![\mathsf{sample}()\ \textbf{every}\ 60{*}10\ \textbf{expire}\ 60{*}60{*}24{*}7]\!](\ldots)$. The machine pushes expression $60 * 60 * 24 * 7$, expression $60 * 10$, and string $"\mathsf{sample}"$; sets up a timed call, and pushes a new empty module onto the operands stack. Let $\mathcal{E}_0 =$

$\langle \mathcal{M}_0, \mathcal{M}_1, \{\}, \mathcal{M}_1, \{\}\rangle$.

$$
\begin{array}{l}
\langle 13, \mathcal{M}_1, \{\}, (13, \mathcal{E}_0, \epsilon, \langle \ldots, \texttt{loadc}\ 6, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 14, \mathcal{M}_1, \{\}, (14, \mathcal{E}_0, 7, \langle \ldots, \texttt{loadc}\ 5, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 15, \mathcal{M}_1, \{\}, (15, \mathcal{E}_0, 24 : 7, \langle \ldots, \texttt{mul}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 16, \mathcal{M}_1, \{\}, (16, \mathcal{E}_0, 168, \langle \ldots, \texttt{loadc}\ 3, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 17, \mathcal{M}_1, \{\}, (17, \mathcal{E}_0, 60 : 168, \langle \ldots, \texttt{mul}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 18, \mathcal{M}_1, \{\}, (18, \mathcal{E}_0, 10080, \langle \ldots, \texttt{loadc}\ 3, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 19, \mathcal{M}_1, \{\}, (19, \mathcal{E}_0, 60 : 10080, \langle \ldots, \texttt{mul}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 20, \mathcal{M}_1, \{\}, (20, \mathcal{E}_0, 604800, \langle \ldots, \texttt{loadc}\ 4, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 21, \mathcal{M}_1, \{\}, (21, \mathcal{E}_0, 10 : 604800, \langle \ldots, \texttt{loadc}\ 3, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 22, \mathcal{M}_1, \{\}, (22, \mathcal{E}_0, 60 : 10 : 604800, \langle \ldots, \texttt{mul}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 23, \mathcal{M}_1, \{\}, (23, \mathcal{E}_0, 600 : 604800, \langle \ldots, \texttt{loadc}\ 0, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 24, \mathcal{M}_1, \{\}, (24, \mathcal{E}_0, \text{"}\mathsf{sample}\text{"} : 600 : 604800, \langle \ldots, \texttt{timer}, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 25, \mathcal{M}_1, \{(\mathit{sample}(), 600, 624, 604824)\}, (25, \mathcal{E}_0, \epsilon, \langle \ldots, \texttt{loadm}\ 2, \ldots\rangle, \mathcal{U}), \epsilon\rangle^{\epsilon}_{\epsilon}
\end{array}
$$

Afterwards, CVM sits idle until the event is triggered and the running call-frame $\mathcal{C}_0$ is replaced.

$$
\begin{array}{l}
\langle 26, \mathcal{M}_1, \{(\mathit{sample}(), 600, 624, 604824)\}, \mathcal{C}_0, \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\ldots \\
\langle 624, \mathcal{M}_1, \{(\mathit{sample}(), 600, 1224, 604824)\}, \mathcal{C}_0, \mathit{sample}()\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\langle 624, \mathcal{M}_1, \{(\mathit{sample}(), 600, 1224, 604824)\}, (0, \epsilon, \epsilon, \mathcal{B}_{\mathsf{next}}, \langle \text{"}\mathsf{sample}\text{"}\rangle), \epsilon\rangle^{\epsilon}_{\epsilon} \rightarrow \\
\ldots \\
\text{where } \mathcal{C}_0 = (26, \mathcal{E}_0, \{\}, \langle \ldots, \texttt{return}\rangle, \mathcal{U}) \\
\text{and } \mathcal{B}_{\mathsf{next}} = \langle \texttt{loadb}, \texttt{loadc}\ 0, \texttt{call}, \texttt{return}\rangle
\end{array}
$$

The translation we present here is simple and there is room for optimisations, but this is not the focus of this paper.

3.3 Run-Time Soundness

We have proved that the specification of CVM given in section 3.2 preserves the semantics of Callas programs. More precisely, we proved that, if we start with a byte-code program $[\![e]\!]$ that is the result of the translation 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 $[\![e']\!]$, which is again the translation of some 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 the following.

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

(sketch) We take e and proceed by induction on the translation of e, analysing the last rule applied from the translation function (*vide* Figures 15, 16, 17 and 18). For each case we consider the result of translating e into byte-code program $[\![e]\!]$; next we use the semantic rules from Figures 19 and 20 to compute $[\![e']\!]$ such that it is exactly the result of translating Callas program e'. Using the rules in Figure 11 we prove each case, namely that $e \rightarrow e'$. For the cases of the **let** $x = e_1$ **in** e_2 and **if** b **then** e_1 **else** e_2 instructions we need to use the induction hypothesis, since for such cases the function is used recursively to translate programs e_1 and e_2.

This theorem establishes a deep link between the operational semantics of the Callas programming language and the operational semantics for the run-time. The importance of this result cannot be overstated. It implies that byte-code programs generated by the Callas compiler (and thus free of run-time protocol errors) is going to be executed by the run-time system in accordance with the operational semantics for the language as given in Section 2. Thus, the byte-code programs will be also immune to misbehaviour on the part of the run-time system. In this way another potentially problematic source of (run-time) errors is eliminated *a priori* by carefully designing the language and its run-time system.

4 PROTOTYPE IMPLEMENTATION AND DEPLOYMENT

As a proof-of-concept we have built two Software Development Kits (SDKs) for programming WSN with Callas in two different platforms, one for real life Sun SPOT devices, and another for the VisualSense simulator. Since we abstract the run-time system (*e.g.*, sensor's hardware) with CVM, the same compiler can be used in the various SDKs. Each SDK, however, must provide its implementation of CVM.

The Callas compiler implementation is divided into three components: a syntactic analyser, a semantic analyser, and a code generator. The compiler uses the syntactic analyser to parse the source code and construct an abstract syntactic tree—an abstract representation of the program based on the syntax in Figure 6. Afterwards, the compiler verifies the parsed program with the semantic analyser, according to our type system (not shown; refer to [11], Figures 4 and 5). Finally, the compiler generates a byte-code program from the abstract representation of the program, following the translation function in Figure 15. To give some flavour 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.

Both SDKs target a Java run-time system (JVM); Sun SPOT applications run on top of Squawk [14], a JVM for embedded systems; and the simulated components (*e.g.*, sensors) in VisualSense are Java classes themselves that run on the standard JVM, which also runs the simulator itself. To improve maintenance both SDKs share most of the code for CVM implementation. Each Java-based SDK must only implement the network component and the available external operations (**extern**). The shared codebase is divided into two parts: the *interpreter* and the *byte-code manipulation*. The former consists of data-structures representing the state of the virtual machine, and implements the reduction rules presented in Section 3.2. 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 byte-code manipulation part includes a parser that loads a byte-code program to create the run-time data-structures needed to start the interpreter, and is also responsible for marshaling and un-marshaling Callas values (for network communication).

The code specific for implementing CVM in Sun SPOT devices is about 400 lines of

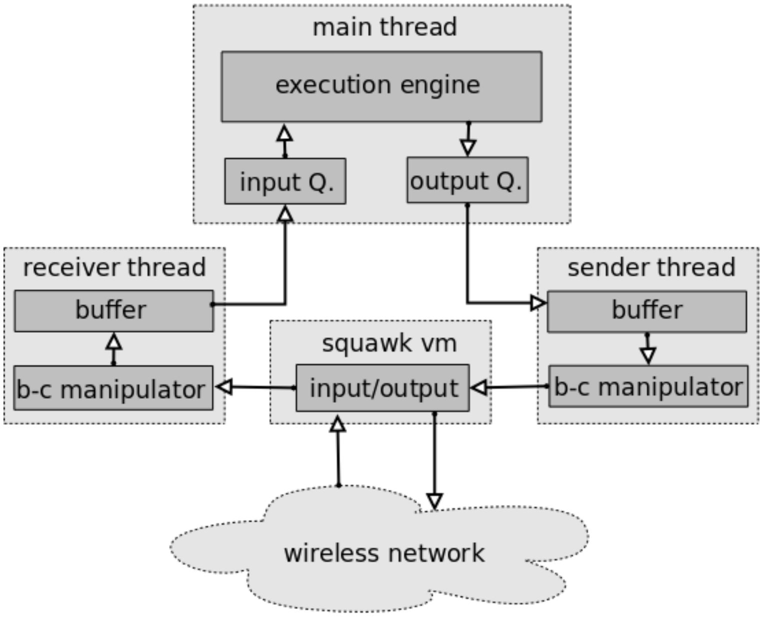

Figure 21: The Callas run-time.

Java code. The network layer is responsible for (un-)marshaling and broadcasting (receiving) remote function calls. We expose external calls that give access to the existing sensors and actuators of Sun SPOTs. There are sensors for luminosity, temperature, motion, orientation, and battery level. As for actuators, we have light-emitting diodes.

The architecture of the run-time system can be seen in Figure 21. 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 data uses the network infrastructure to accept byte-code that is un-marshaled into function calls, subsequently consumed by the main thread. Finally, the thread transmitting data to the network consumes available function calls, produced by the main thread, and marshals them into byte-code that is broadcast to the network. 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.

The SDK for simulating Callas applications using VisualSense consists on two applications (besides the Callas compiler): a generator of simulation models and the run-time system, VisualSense extended to run Callas applications. Our compiler currently uses a file format, called the *network file*, to describe the code that runs on a certain category of sensors, the existing remote function calls, and the available external operations. 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 emit a simulation model, to be

Nodes	Duration (hours)	Memory Usage (GB)
100	0.05	0.25
200	0.21	0.23
400	0.48	0.26
500	0.72	0.31
1000	1.43	0.45
2000	3.80	0.87
3000	6.65	1.44
4000	9.97	2.14
5000	15.90	2.55

Table 1: Results for networks with sizes from 100 up to 5000 nodes.

used by VisualSense. In VisualSense, adding specific support for simulated devices running CVM requires an implementation work similar to the one performed for Sun SPOTs, namely, writing the network-related code, the external operations of the run-time system, and the intermediary code that executes CVM in the simulator. The component that exposes CVM as a simulation element amounts to 900 lines of code. We performed a simulation of the running example (Figures 3 and 4) for 10 minutes, but varied the amount of devices in the network, as depicted in Table 1. The results were obtained with VisualSense 7.01 on a Linux based PC with an Intel QuadCore 2.66GHz 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 node 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 nodes.

5 CONCLUSION AND FUTURE WORK

In this paper we address the problem of providing WSN with programming languages that minimise or even eliminate some types of run-time 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 run-time systems, to be *safe-by-design*. Accordingly, we design a language that enjoys type-safety. Well-typed programs in this language cannot produce run-time errors due to misuse of component interfaces. Another design principle we adopt is the specification of the language run-time system in such a way that we can verify its soundness, *i.e.,* that it preserves the operational semantics of the language. Programs executed by this run-time system never misbehaves. A further design principle, which is the subject of ongoing work, allows the proof that the language compiler preserves the semantics of the programming language. 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 run-time errors.

To demonstrate the feasibility of this approach we present a new programming language, Callas, and show how it can be formalised in a precise way. Such a language, we have shown, is type-safe. Moreover, we provide a specification for the run-time and the compilation scheme for the language. We use these definitions to prove that the run-time specification preserves the language semantics.

We describe two SDKs for programming and deploying WSN applications, one targets real life Sun SPOT devices, and the other is directed at simulated environments (via VisualSense). We detail these two implementations of the language in the form of a compiler and two run-time system (abstracted by CVM) that follow the formal definitions above.

In terms of future work, besides the ongoing work on the Callas compiler, we are interested in developing higher level idioms for programming WSN (e.g. a macroprogramming language) fully encoded in Callas. With the idea of making the run-time system safer, we intend to incorporate types into CVM, making it reject programs that misbehave. Afterwards, we can then use type-preserving compilation to retain the operational properties captured by types from end-to-end.

5.1 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] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91)*, pages 131–142. The ACM Press, 1991.

[6] 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.

[7] 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.

[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 JoÃ£o 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. *Electronic Proceedings in Theoretical Computer Science*, 17:49–62, 2010.

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

[13] 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.

[14] 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.

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