A Calculus for Programming Wireless Sensor Networks

Luís Lopes*, Francisco Martins†, Miguel S. Silva*, João Barros‡

*Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto, Portugal
†Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Portugal
‡Instituto de Telecomunicações (IT), DCC-FCUP, Porto, Portugal
lblopes@dcc.fc.up.pt

***Abstract*—We present a process calculus that models the communication and computational aspects of sensor network applications. The calculus focuses on a basic set of primitives for programming sensor networks that support code deployment, communication and local processing, and provides a very expressive core-language. The calculus also provides the tools to verify the robustness of sensor network applications and protocols.**

keywords: Sensor Networks, Ad-Hoc Networks, Ubiquitous Computing, Process-Calculi, Programming Languages.

I. Introduction

The ability to program large-scale networks of power-constrained and computationally restricted sensing devices [3] is widely recognized as one of the key challenges towards turning the sensor network vision into reality. In most of the current applications [4], the sensor nodes are controlled by module-based operating systems such as TinyOS [1] and are programmed using somewhat ad-hoc languages *e.g.* nesC [8] or TinyScript/Maté [11]. Recent middleware developments such as Deluge [10] and Agilla [7], and programming languages and enviroments such as Regiment [18] and EnviroSuite [13], provide higher level programming abstractions including massive code deployment where needed.

Despite these undeniably useful practical developments, it is fair to state that current systems fall short of providing a fundamental formal model that captures the computational and communication properties of sensor networks. Once fully developed, such a model could serve as the basis for the development of robust and provably correct implementations of high-level programming languages for sensor hardware platforms.

Our main contribution is a programming model with stronger formal support and analytical capabilities than the aforementioned practical solutions. Given the inherent distributed and concurrent nature of sensor network operations, we choose to base our model for wireless ad-hoc sensor networks on the mature formalism of process calculi [9, 17]. The theory developed for process calculi is very rich and expressive, providing tools with which sensor network applications and protocols can be proven correct with respect to a given specification, *e.g.* using *bissimulation* techniques [22]. The theory also provides the means to thoroughly quantify the resource usage by sensors when running applications, *e.g.* using *type systems* to estimate the run-time data flow. Some work exists on modeling wireless systems with process calculi but it focuses on the protocol layer of the networks and does not address the peculiarities of programming *ad-hoc* sensor networks [15, 19, 21].

The calculus we introduce, aptly called Calculus for Sensor Networks (CSN), focuses on a basic set of primitives for programming sensor networks that support (a) massive code deployment, (b) wireless communication among nodes, and (c) distributed processing based on local information [12, 24]. The corresponding core-language is very expressive and adheres to the *global view* advocated for developing sensor network applications. Since we focus entirely on the challenges of programming sensor network applications, our model abstracts away from the link layer of the protocol stack, *i.e.* we assume that programmers do not have to deal with transmission errors and packet losses.[1]

The rest of the paper is organized as follows. Section II describes our calculus for sensor networks by specifying the syntax and the semantics. Its expressiveness is exemplified in Section III. Section IV is then devoted to an outline of the underlying type system and the paper concludes with Section V.

II. The Calculus

We start by describing the proposed calculus, whose syntax is provided by the grammar shown in Figure 1, and whose operational semantics results from the congruence and reduction relations listed in Figures 2 and 3.

Syntax. A CSN program is composed by a set of top-level global definitions D for sensors and a specification of a network. The latter is a flat, unstructured collection of sensors S combined using the parallel composition operator. The sensors are assumed to be immersed in a (scalar or vector) field and to

This research was partly supported by the Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) under grant POSC/EIA/62199/2004.

[1]Notice that the word "calculus" is used here strictly in the sense of a *deterministic* formalism for developing operating systems and programming languages. Consequently, our work has practically *no* relation to the body of work in communications engineering that focuses on the development of a *stochastic* calculus for modeling sensor networks (see *e.g.* [23] and references therein).

F	::=	*Programs*
	$D\ S$	defs. & sensors
D	::=	*Definitions*
	ε	empty definition
	$\mid\ x = v$	constant
S	::=	*Sensors*
	off	offline
	$\mid\ [\vec{P} :: M]^p_e$	sensor
	$\mid\ S \mid S$	composition
P	::=	*Processes*
	v	value
	$\mid\ v.l(\vec{v})$	function call
	$\mid\ v.\,\textbf{install}\ v$	module update
	$\mid\ \textbf{let}\ x = P\ \textbf{in}\ P$	local variable
M	::=	*Modules*
	$\{l_i = (\vec{x}_i)\,P_i\}_{i\in I}$	module interface
v	::=	*Values*
	b	built-in value
	$\mid\ x$	variable
	$\mid\ \textbf{net}$	broadcast
	$\mid\ \textbf{loc}$	top-level module
	$\mid\ M$	module

Fig. 1. The syntax of CSN.

be able to measure its intensity at their positions in space. The field describes the distribution of some physical quantity we want to model (*e.g.* temperature, pressure, humidity) in space. The position of a sensor is given in some coordinate system.

A sensor $[\vec{P} :: M]^p_e$ represents an abstraction of a physical sensing device running a sequence of processes $\vec{P}$ and with a memory M (code plus state). Module M is a collection of functions (the interface) that the sensor makes available for internal and for external usage. Each function, $l = (\vec{x})P$, is identified by label l and defined by an abstraction $(\vec{x})P$: a process P with parameters $\vec{x}$. Intuitively, the collection of functions of a sensor may be interpreted as the function calls of some "tiny" operating system installed in the sensor at boot time or the functionalities dynamically uploaded to the sensor. We assume the programmer is able to query the sensors for their position (p), energy supply (e), and field readings, using built-in operating system calls.

Processes are ranged over by P. A function call, $v.l(\vec{v})$, calls the function l (with arguments $\vec{v}$) in some value v. Value v may be an anonymous module, the top level module, if the target is **loc**, or the *broadcast address*, if the target is **net**. In the last case, the call is broadcasted to the network neighborhood of the sensor. Installing or replacing functions in a module can be done with the construct $v.\,\textbf{install}\ v'$, which adds the functions in v' to the module v, eventually replacing existing implementations with new ones. In particular, this construct allows the state of modules to be modelled. The **let** construct allows processes to create local variables to hold intermediate values in computations. A sequential composition construct (;) can be easily obtained as syntactic sugar from the **let** construct and we use it to impose a more imperative style of programming. Values are the data exchanged between sensors and comprise basic values that can intuitively be seen as the primitive data types supported by the sensor's hardware, and modules that are constructed dynamically.

$$S_1 \mid S_2 \equiv S_2 \mid S_1 \qquad S \mid \textbf{off} \equiv S$$
$$S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3 \qquad \text{(S-MONOID-SENSOR)}$$
$$[\vec{P} :: M]^p_e \equiv [\vec{P} :: M]^p_e\{\textbf{off}\} \qquad \frac{e < \min(\mathsf{e_{in}}, \mathsf{e_{out}})}{[\vec{P} :: M]^p_e \equiv \textbf{off}}$$
(S-BROADCAST, S-BAT-EXHAUSTED)

Fig. 2. Structural congruence for processes and sensors.

A Simple Example. We start with a very simple ping program. We denote as Ping and Sink the modules installed in any of the anonymous sensors in the network and in the sink, respectively. The Ping module has a ping function that gets the MAC address for the device, forwards it to the network, and propagates the original ping call. The forward function in the sink just logs any incomming MAC address. So, the overall result of the call **net**.ping() in the sink is that all reachable sensors in the network will receive this call and will flood the network with their MAC addresses. These values eventually reach the sink and get logged.

```
Ping={
   ping=()let m=loc.mac_addr() in
          net.forward(m);
          net.ping()
   forward=(x)net.forward(x)
}
Sink={forward=(x)log_mac(x)}
[net.ping()::Sink]|[{}::Ping]|...|[{}::Ping]
```

Semantics. Following the tradition in process calculi we present the reduction relation with the help of a structural congruence relation [16].

The structural congruence relation $\equiv$, depicted in Figure 2, is defined as the smallest congruence relation on sensors closed under those rules. The parallel composition of sensors is commutative and associative with **off** as the neutral element (*vide* Rule S-MONOID-SENSOR). Rule S-BROADCAST allows for a sensor to start the broadcasting operation. An offline sensor is one with insufficient battery level for performing an internal or an external reduction step (*vide* Rule S-BAT-EXHAUSTED).

The reduction relation on networks, notation $S \rightarrow S'$, describes how a sensor S can evolve (reduce) to sensor S'. Reduction in a sensor occurs at the head of a sequence $\vec{P}$. In other words, in a sequence $P, \vec{P}$, process P is running and those in $\vec{P}$ are waiting in a queue. We also allow for reduction within the **let** construct. Thus the P in the example above can be of the form $\textbf{let}\ \ x = P'\ \textbf{in}\ \ P''$ and we allow the reduction *in situ* of P'. For this reason we present the relation using reduction contexts, or places were reduction may occur. These contexts, denoted $\mathcal{C}[\![]\!]$, are defined as follows: $\mathcal{C}[\![]\!] \ ::= \ [\,] \ \mid \ \textbf{let}\ \ x = [\,]\ \textbf{in}\ \ P$. Thus, $\mathcal{C}[\![P]\!]$ denotes the process P inserted in the $[\,]$ hole of any of the above contexts.

$$\frac{M(l) = (\vec{x})P \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathbf{loc}\,.l(\vec{v})]\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![P[\vec{v}/\vec{x}]]\!], \vec{P} :: M]^p_{e'}} \text{(R-FUNCTION-TOP)}$$

$$\frac{l \notin \mathrm{dom}(M)}{[\mathcal{C}[\![\mathbf{loc}\,.l(\vec{v})]\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![\mathbf{loc}\,.l(\vec{v})]\!], \vec{P} :: M]^p_{e'}} \text{(R-NO-FUNCTION-TOP)}$$

$$\frac{M'(l) = (\vec{x})P \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![M'.l(\vec{v})]\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![P[\vec{v}/\vec{x}]]\!], \vec{P} :: M]^p_{e'}} \text{(R-FUNCTION)}$$

$$\frac{\mathrm{withinRange}(p_1, e_1, p_2) \qquad e_1 \geq \mathsf{e_{out}}}{[\mathcal{C}[\![\mathbf{net}\,.l_i(\vec{v})]\!], \vec{P} :: M]^{p_1}_{e_1}\{S\} \mid [\vec{P}' :: M']^{p_2}_{e_2} \rightarrow [\mathcal{C}[\![\mathbf{net}\,.l_i(\vec{v})]\!], \vec{P} :: M]^{p_1}_{e'_1}\{S \mid [\vec{P}', \mathbf{loc}\,.l_i(\vec{v}) :: M']^{p_2}_{e'_2}\}} \text{(R-BROADCAST)}$$

$$[\mathcal{C}[\![\mathbf{net}\,.l_i(\vec{v})]\!], \vec{P} :: M]^p_e\{S\} \rightarrow [\mathcal{C}[\![\{\}]\!], \vec{P} :: M]^p_{e'} \mid S \quad \text{(R-RELEASE)}$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathbf{loc}\,.\,\mathbf{install}\ M']\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![M + M']\!], \vec{P} :: M + M']^p_{e'}} \text{(R-INSTALL-TOP)}$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![M'.\,\mathbf{install}\ M'']\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![M' + M'']\!], \vec{P} :: M]^p_{e'}} \text{(R-INSTALL)}$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\mathbf{let}\ x = v\ \mathbf{in}\ P]\!], \vec{P} :: M]^p_e \rightarrow [\mathcal{C}[\![P[v/x]]\!], \vec{P} :: M]^p_{e'}} \text{(R-LET)}$$

$$[P, \vec{P} :: M]^p_e \rightarrow [\vec{P}, P :: M]^p_{e'} \qquad \frac{S \rightarrow S'}{S \mid S'' \rightarrow S' \mid S''} \text{(R-SWITCH, R-NETWORK)}$$

$$\frac{S_1 \equiv S_2 \qquad S_2 \rightarrow S_3 \qquad S_3 \equiv S_4}{S_1 \rightarrow S_4} \text{(R-CONGR)}$$

Fig. 3. Reduction semantics for sensors.

The reduction relation is inductively defined by the rules in Figure 3 and is parametric on two constants $\mathsf{e_{in}}$ and $\mathsf{e_{out}}$ that represent the amount of energy consumed when performing internal computation steps ($\mathsf{e_{in}}$) and when broadcasting messages ($\mathsf{e_{out}}$). This is a very simple power checking mechanism that regulates the sensor behavior. Although this is not our main focus at this point, such accounting of the sensor resources can be much refined not only for power management but also to control memory and processor usage, by adding more complexity to the preconditions and to the transitions in these rules.

Process P in a sensor $[P, \vec{P} :: M]^p_e$ may: (a) call functions in the top-level module M (Rules R-FUNCTION-TOP and R-NO-FUNCTION-TOP), in anonymous modules (Rule R-FUNCTION), and in the network (Rules R-BROADCAST and R-RELEASE); (b) install new functions in the top-level module M (Rule R-INSTALL-TOP), and in modules (Rule R-INSTALL); (c) compute intermediate values and assign them to new variables (Rule R-LET), and (d) stop and allow another process to run (Rule R-SWITCH).

A call to a local function l with arguments $\vec{v}$ in a module M, be it the top level module or an anonymous one, such that $M(l) = (\vec{x})P$, results in the process P where the variables in $\vec{x}$ are replaced with the values $\vec{v}$. Traditionally, typed programming languages use a type system to ensure that there are no calls to undefined functions, ruling out all other programs at compile time. Our approach allows an extra degree of flexibility. When function l is not present in module M, the reduction depends on whether M is the top level module or not. In the first case, we have decided to keep the call active (see Rule R-NO-FUNCTION-TOP). In the second case, calling an undefined function in an anonymous module causes the program to get *stuck*; such a program would be ruled out by our type system (see Section IV). We envision that if we call a function in the network after some code has been deployed (see Section III), some sensors may receive the function call before the code is actually deployed. With the semantics we propose, the call waits for the code to be installed.

Sensors communicate with the network by broadcasting function calls on the top-level modules of other sensors. During a broadcast operation the message must only reach each neighborhood sensor once. Notice that the same message can reach the same sensor multiple times, albeit only as the result of the echoing of the message in subsequent broadcast operations. We model the broadcasting of messages in two stages, abstracting away from the metric used to establish the connectivity between any two sensors. Rule R-BROADCAST calls function l in the remote sensor if the receiving sensor is *within range* of the broadcasting sensor. Several metrics may be chosen, *e.g.* based on the geometric distance between the sensors or on the received power at the destination node. Here, we instantiate the metric based on the positions of the sensors and on the battery power of the broadcasting sensor. Each sensor that receives the call is put in the membrane associated with the broadcasting sensor, thus preventing multiple deliveries of the same message in the same broadcast. Observe that the rule does not enforce the interaction with all sensors in the neighborhood of the broadcasting sensor. Rule R-RELEASE finishes the broadcast by consuming the operation ($\mathbf{net}\,.l(\vec{v})$), and dissolves the membrane. A broadcast operation starts with the application of Rule S-BROADCAST, proceeds with multiple (eventually none) applications of Rule R-BROADCAST (one for each target sensor), and terminates with the application of Rule R-RELEASE.

Installing a set of functions M' in an existing module M, be it at top level or not, amounts to adding to M the functions in M' (absent in M), and to replace (in M) the functions common to both M and M'. Rigorously, the operation of installing the functions M' on top of M, denoted $M + M'$, may be defined as $M + M' = (M \setminus M') \cup M'$. The $+$ operator is reminiscent of Abadi and Cardelli's operator for updating functions in their imperative object calculus [2].

A running process P in a sequence $P, \vec{P}$, may stop its execution at any time and allow the execution of the next process in the sequence $\vec{P}$. Process P is placed at the end of the sequence, actually implementing a very simple *round-robin* interleaved execution model. This may be seen as a very simple scheduling mechanism provided by an underlying operating system and serves essentially the purpose of ensuring that sensors eventually process incoming network communication. Another solution would be to let the programmer yield the cpu to other processes, for example using a construct like **post** P [1]. Finally, Rule R-PARALLEL allows reduction to happen in networks of sensors, and Rule R-STRUCTURAL brings structural congruence into the reduction relation.

An example of reduction. To illustrate the operational se-

mantics of CSN, we present the reduction steps for the ping example discussed above. Due to space constraints we consider a rather simple network with just the sink and another sensor. To simplify the notation we assume that the positions and energy levels of the sink and the sensor are, respectively, p and e, and p' and e'.

```
[net.ping():: Sink] | [{}:: Ping]
```

We assume that the sensor is within range from the sink and vice-versa. This network may reduce as follows:

[net.ping ():: Sink] | [{}:: Ping] $\equiv$ (S-BROADCAST)

[net.ping ():: Sink]{off} | [{}:: Ping] $\rightarrow\equiv$ (withinRange(p, e, p'), R-BROADCAST, S-MONOID-SENSOR)

[net.ping ():: Sink] {[{},loc.ping ():: Ping]} $\rightarrow\equiv$ (R-RELEASE, R-SWITCH)

[{}:: Sink]|[loc.ping (),{}:: Ping] $\rightarrow$ (R-FUNCTION-TOP)

[{}:: Sink]|[let m=loc.addr() in net.forward(m);net.ping (),{}:: Ping] $\equiv$ (R-LET, S-MONOID-SENSOR)

[net.forward(m);net.ping (),{}:: Ping]|[{}:: Sink] $\rightarrow$ (S-BROADCAST)

[net.forward(m);net.ping (),{}:: Ping]{off} | [{}:: Sink] $\rightarrow\equiv$ (withinRange(p', e', p), R-BROADCAST, S-MONOID-SENSOR)

[net.forward(m);net.ping (),{}:: Ping]{[{},loc.forward(m)::Sink]} $\rightarrow$ (R-RELEASE, R-SWITCH)

[net.ping (),{},{}:: Ping] | [{},loc.forward(m)::Sink] $\equiv$ (S-BROADCAST)

[net.ping (),{},{}:: Ping]{off} | [{},loc.forward(m)::Sink] $\equiv$ (R-RELEASE, R-SWITCH)

[{},{},{}:: Ping] | [loc.forward(m),{}::Sink] $\equiv$ (R-FUNCTION-TOP)

[{},{},{}:: Ping] | [log_mac(m),{}::Sink]

After these reduction steps the sink gets the MAC address m from the sensor and logs it. The sensor is idle waiting for further interaction. A sequence of processes {}, {}, ...,{} can be garbage collected into a single (value) process {}. So, sensor [{},{},{}:: Ping] becomes [{}:: Ping].

III. Examples

In the following programming examples we assume that the network layer handles *scoped flooding*, which can easily be implemented with CSN although the code is somewhat lengthy to be presented here.

Bootstrapping the Network. The *ping* example from section II assumes that we have some means of *bootstrapping* the module Ping to the sensor nodes.

```
Boot={deploy=(x)loc.install x;net.deploy(x)}
Ping={...}
Sink={forward=(x)log_mac(x)}
[net.deploy(Ping);net.ping():: Sink]|
[{}:: Boot]|...|[{}:: Boot]
```

Here we call a deploy function at boot time, from a Boot module within the sensors (think of it as a basic OS module). The Ping module is sent as a parameter of the call. Function deploy takes the module given as argument and installs it locally, while propagating the call to the network neighborhood. Once deployed, the code is activated with a call to ping from the sink as above.

Continuous Sampling. In this example the sensors continuously take field readings and report them to the sink for logging purposes.

```
Boot={...}
Sample={
  sample=()net.sample();
           loc.install {
             sample=() let f=loc.field() in
                       net.forward(f);
                       loc.sample()
           };
           loc.sample()
  forward=(x)net.forward(x)
}
Sink={forward=(x)log_field(x)}
[net.deploy(Sample);net.sample():: Sink]|
[{}:: Boot]|...|[{}:: Boot]
```

We program the application from the bootstrap point. After the Sample module has been uploaded to the sensors, the sink just calls the function sample to begin the sampling. The first call to sample in any sensor starts by propagating the call to the network neighborhood; then the sample function changes itself through an **install** call and finally calls the new code to start the sampling cycle. The new code for sample reads values from the field, forwards the results to the network, and calls itself again to implement a cycle.

Reading the Maximum of the Field. This example computes the maximum field value read by any sensor in a given network. A Max module is uploaded to each sensor that has two functions.

```
Boot={...}
Max={
  max=()let v=loc.field() in
        loc.install {max=()v};
        net.forward(v);
        v
  forward=(x)let v=loc.max() in
             if x > v then
               net.forward(x);
               loc.install {max=()v}
}
Sink={
  max=()0.0
  forward=(x)let v=loc.max() in
             if x > v then
               loc.install {max=()x};
             {}
}
[net.deploy(Max);net.max(),Sink]|
[{},Boot]|...|[{},Boot]
```

The first time max is called (from the sink) in a sensor, it reads the field value v, changes its own code to max=()v, and forwards the value to the network. In fact, after sampling the field, function max acts as an *attribute* of the sensor since it just returns a value. Here, the value is a floating point number (a primitive type), but in other examples attributes can hold arbitrarily complex modules. A sensor only forwards values from incoming forward calls if the former are greater than the local maximum kept in max. In this case, max gets updated too. This way the network implements a sort of dynamic filter that minimizes the number of forward calls. The sink starts off with

a 0.0 value (here we assume that the field readings must be positive floating point numbers) and uses the incoming forward calls to update the value in max.

Secure Code Deployment. In the previous examples the code deployment is inherently insecure. In fact, any sensor in the network or even external entities might send deploy calls to the network and install modules unchecked. Here we show how we can prevent this problem assuming that the sink and the remainder of the sensors share a secret *key*. The distribution of the keys is assumed to be done in some secure scheme for wireless sensor networks, *e.g.* trusted third party [20], public key [14] or key pre-distribution [5, 6, 25].

```
Boot={
   sdeploy=(x)let k = loc.secret() in
              let y = loc.decrypt(k,x)
              in loc.install y;
              net.sdeploy(x)
}
Ping={...}
Sink={forward=(x)log_mac(x)}
[let k=loc.secret() in
 net.sdeploy(loc.encrypt(k,Ping));
 net.ping()::Sink]|
[{}::Boot]|...|[{}::Boot]
```

The Boot module implements a *secure deploy* function, sdeploy, that receives one argument. This argument is assumed to be a module issued from the sink and encrypted using the shared secret key, k. The function attempts to decrypt the argument using the key k and the resulting module is installed locally. However, if the sender is not the sink, decrypt returns an empty module which, although installed, does not alter the code in the sensor. The sink simply encrypts the Ping module (in this example), deploys it to the network and triggers its execution.

IV. The Type System

Types T (depicted in Figure 4) are built from the built-in type B (*e.g.* battery level, position, or received power) and from the broadcast type **Net**, using the constructor for module types $\{l_i\colon \vec{T_i} \to T_i\}_{i\in I}$ (and $[l_i\colon \vec{T_i} \to T_i]_{i\in I}$). Type **Net** is assigned to value **net** and distinguishes local function calls from broadcast communications (via remote function call). A type $\{l_i\colon \vec{T_i} \to T_i\}_{i\in I}$ describes a module represented as a collection of (distinctly named) functions. Each function l_i has type $\vec{T_i} \to T_i$, where $\vec{T_i}$ is the type of the parameters of the function and T_i is its return type. For instance, type $\{\mathsf{ping}\colon \epsilon \to \{\}, \mathsf{forward}\colon B \to \{\}\}$ is the type of the module Ping presented in Section II. Function ping has no parameters and returns an empty module (the result from the final **net**.ping() operation). Function forward accepts a built-in value and returns $\{\}$, like ping. Type $[l_i\colon \vec{T_i} \to T_i]_{i\in I}$ denotes the type of the sensor's top-level module. It plays an important role when typing code installation. When referring to both types of modules we use notation $< l_i\colon \vec{T_i} \to T_i >_{i\in I}$.

A typing Γ is a partial function of finite domain from variables to types. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathrm{dom}(\Gamma)$ we write $\Gamma, x\colon T$ for the disjoint union of Γ with $x\colon T$.

T ::=	*Types*
B	built-ins
| **Net**	broadcast
| $\{l_i\colon \vec{T_i} \to T_i\}_{i\in I}$	module
| $[l_i\colon \vec{T_i} \to T_i]_{i\in I}$	top-level module

Fig. 4. The syntax of types.

$$\Gamma \vdash b\colon B \qquad \Gamma, x\colon T \vdash x\colon T \qquad \text{(T-BUILT-IN, T-VAR)}$$

$$\frac{\forall i.\Gamma \vdash x_i\colon T_i}{\Gamma \vdash \vec{x}\colon \vec{T}} \qquad \Gamma \vdash \textbf{net}\colon \textbf{Net} \qquad \text{(T-SEQ, T-NET)}$$

$$\Gamma, \textbf{loc}\colon [l_i\colon \vec{T_i} \to T_i]_{i\in I} \vdash \textbf{loc}\colon [l_i\colon \vec{T_i} \to T_i]_{i\in I} \qquad \text{(T-NET, T-LOC)}$$

$$\frac{\forall i \in I.\Gamma, \vec{x_i}\colon \vec{T_i} \vdash P_i\colon T_i}{\Gamma \vdash \{l_i = (\vec{x_i})\, P_i\}_{i\in I}\colon \{l_i\colon \vec{T_i} \to T_i\}_{i\in I}} \qquad \text{(T-OBJ)}$$

Fig. 5. Typing rules for values.

Type judgements are of three forms: $\Gamma \vdash v\colon T$ means that value v has type T, under the assumptions in typing Γ; $\Gamma \vdash P\colon T$ asserts that process P has type T, under the assumptions in Γ; and $\Gamma \vdash S$ means that sensor network S is well typed, assuming the typing Γ.

The rules for typing values (Figure 5) are straightforward. We comment on rule T-LOC that assigns type $[l_i\colon \vec{T_i} \to T_i]_{i\in I}$ to the (special) variable **loc**. It represents the interface of all sensors and is invariant during type checking. This means that the interface for sensors is fixed by the application programmer, before type checking takes place.

As for processes, function calls are separated in local calls (Rule T-CALL) and remote calls (Rule T-BCAST). In a local call, function l_j must be part of the target module ($j \in I$), the type of the arguments must agree with the type of the parameters ($\vec{v}\colon \vec{T_j}$), and the type of the invocation (T_j) is the return type of the function. A remote call ($v\colon$ **Net**) is type checked as a local call, apart from its return type that is always the empty module ($\{\}$), meaning that the return value of a remote call is ignored. Code installation (Rule T-INST) is allowed either in anonymous modules, or in the sensor's top-level module. The definition of operation $T_1 \oplus T_2$ is similar to that of operation $+$ for combining modules, but has only a meaning for $[\_] \oplus [\_]$, $[\_] \oplus \{\_\}$, and $\{\_\} \oplus \{\_\}$. Allowing $\{\_\} \oplus [\_]$ may cause an anonymous module to refer to undefined functions, since it inherits the complete interface from the type of the top-level module. The result of an **install** operation is the altered module.

Regarding the typing rules for sensors, we focus on rule T-SENSOR. When typing a sensor we make sure that the functions available in the sensor's top-level module conform with the global, network-wide, interface defined by **loc**. Notice that a sensor may offer just a subset of the interface functions ($I \subseteq J$), since some of them may not yet be available (installed) in the sensor.

The following result ensures that types are preserved during reduction.

Theorem 1 (Subject Reduction): If $\Gamma \vdash S$, $S \to S'$, then $\Gamma \vdash S'$.

$$\frac{\Gamma \vdash v:\ < l_i : \vec{T_i} \to T_i >_{i \in I} \qquad \Gamma \vdash \vec{v} : \vec{T_j} \quad j \in I}{\Gamma \vdash v.l_j(\vec{v}) : T_j} \quad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash v : \textbf{Net} \qquad \Gamma \vdash \textbf{loc}\,.l(\vec{v}) : \_}{\Gamma \vdash v.l(\vec{v}) : \{\}} \quad \text{(T-BCAST)}$$

$$\frac{\Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2 \qquad T_1 \oplus T_2 \text{ defined}}{\Gamma \vdash v_1\,.\textbf{install}\ v_2 : T_1 \oplus T_2} \quad \text{(T-INST)}$$

$$\frac{\Gamma \vdash P_1 : T_1 \qquad \Gamma, x : T_1 \vdash P_2 : T_2}{\Gamma \vdash \textbf{let}\ x = P_1\ \textbf{in}\ P_2 : T_2} \quad \text{(T-LET)}$$

Fig. 6. Typing rules for processes.

$$\frac{\Gamma \vdash M : \{l_i : \vec{T_i} \to T_i\}_{i \in I} \qquad \Gamma \vdash pe : BB \qquad \Gamma \vdash \textbf{loc} : [l_j : \vec{T_j} \to T_j]_{j \in J} \quad \Gamma \vdash \vec{P} : \_ \quad I \subseteq J}{\Gamma \vdash [\vec{P} :: M]^p_e} \quad \text{(T-SENSOR)}$$

$$\Gamma \vdash \textbf{off} \qquad \frac{\Gamma \vdash [\vec{P} :: M]^p_e \qquad \Gamma \vdash S}{\Gamma \vdash [\vec{P} :: M]^p_e\{S\}} \quad \text{(T-TERM, T-BSENSOR)}$$

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1 \mid S_2} \qquad \frac{\Gamma \vdash P : \_ \qquad \Gamma \vdash \vec{P} : \_}{\Gamma \vdash P, \vec{P} : \_} \quad \text{(T-PAR, T-SEQP)}$$

Fig. 7. Typing rules for sensor networks and for process sequences.

The proof proceeds by induction on the derivation tree for the reduction $S \to S'$ and is a straightforward case analysis; due to space constraints we omit it, as well as the standard intermediate results required for the proof.

V. Conclusions

We presented a programming model for wireless sensor networks based on a process calculus and provided a full specification of its syntax and operational semantics. The calculus provides a core language based on a small yet powerful set of primitive operations, which we deem as expressive and adequate for the goal at hand. We introduced a static type system that enables network applications written in the core language to be checked for protocol errors at compile time, allowing for premature detection of bugs. As a first step to establish the quality of the model we proved a *subject reduction* result.

In terms of practical implementation, we have ready a first prototype of an interpreter for CSN and are now working on a compiler and run-time system that shall enable us to program real-world sensor networks using a CSN-based programming language.

Acknowledgements. The authors gratefully acknowledge insightful discussions with Gerhard Maierbacher (Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto).

References

[1] The TinyOS Documentation Project. Available at `http://www.tinyos.org`.
[2] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *TAPSOFT'95*, number 915 in LNCS, pages 471–485. Springer-Verlag.
[3] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.
[4] D. E. Culler and H. Mulder. Smart Sensors to Network the World. *Scientific American*, 2004.
[5] W. Du, J. Deng, Y. Han, P. Varshney, J. Katz, and A. Khalili. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. *ACM Transactions on Information and System Security*, 8(2):228–258, 2005.
[6] L. Eschenauer and V. Gligor. A Key-Management Scheme for Distributed Sensor Networks. In *CCS'02*, pages 41–47. ACM Press.
[7] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *ICDCS'05*, pages 653–662. IEEE Press.
[8] 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 *PLDI'03*, pages 1–11. ACM Press.
[9] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, number 512 in LNCS, pages 133–147. Springer-Verlag.
[10] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *ENSS'04*, pages 81–94. ACM Press.
[11] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS X*, pages 85–95. ACM Press.
[12] L. Lopes, F. Martins, M. S. Silva, and J. Barros. A Formal Model for Programming Wireless Sensor Networks. Available from `http://arxiv.org`, cs/0702042, February 2007.
[13] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. EnviroSuite: An Environmentally Immersive Programming Framework for Sensor Networks. *ACM TECS*, 5(3):543–576, 2006.
[14] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *SECON'04*. IEEE Press.
[15] N. Mezzetti and D. Sangiorgi. Towards a Calculus for Wireless Systems. In *MFPS '06*, volume 158 of *ENTCS*, pages 331–354. Elsevier Science.
[16] R. Milner. A Calculus of Communicating Systems. Number 92 in LNCS. Springer-Verlag, 1980.
[17] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.
[18] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *DMSN'04 Workshop*.
[19] K. Ostrovský, K. V. S. Prasad, and W. Taha. Towards a Primitive Higher Order Calculus of Broadcasting Systems. In *PPDP'02*, pages 2–13. ACM Press.
[20] A. Perrig, R. Szewczyk, J. Tygar, V. Wen, and D. Culler. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.
[21] K. V. S. Prasad. A Calculus of Broadcasting Systems. In *TAPSOFT'91*, number 493 in LNCS, pages 338–358. Springer-Verlag.
[22] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
[23] Jens Schmitt and Utz Roedig. Sensor Network Calculus - A Framework for Worst Case Analysis. In *DCOSS'05*, number 3560 in LNCS, pages 141–154. Springer-Verlag.
[24] M. S. Silva, F. Martins, L. Lopes, and J. Barros. A Calculus for Sensor Networks. Available from `http://arxiv.org`, cs/0612093, November 2006.
[25] S. Zhu, S. Setia, and S. Jajodia. LEAP: efficient security mechanisms for large-scale distributed sensor networks. In *CCS '03*, pages 62–72. ACM Press.