A Formal Model for Programming Wireless Sensor Networks

Luís Lopes¹, Francisco Martins², Miguel S. Silva¹, and João Barros¹
¹Departamento de Ciência de Computadores, FCUP, Portugal.
²Departamento de Informática, FCUL, Portugal.

Introduction. While there is considerable research activity on programming tools and environments for sensor networks (cf. [2]), we believe that these efforts fall short of providing a fundamental formal model that captures their computational and communication properties. Such a model can serve as the basis for the development of robust and provably correct implementations of high-level programming languages for these hardware platforms.

In this paper we propose a programming model with stronger formal support and analytical capabilities than the above mentioned solutions. Our approach is to model ad-hoc sensor networks with process calculi [1, 3] given the inherent distributed and concurrent nature of sensor network applications. 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. The theory also provides the means to thoroughly quantify the resource usage by sensors when running applications and when using specific protocols. 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 (cf. [2]).

The calculus we introduce, aptly called Calculus for Sensor Networks (CSN), focuses on a basic, yet very expressive set of primitives for programming sensor networks that support code deployment, communication, and local processing. It abstracts away from the link layer of the protocol stack, in other words assuming that the programmer does not have to deal with
transmission errors and packet losses.

The Calculus. We briefly sketch the syntax and semantics of CSN. A detailed account of the calculus may be found in [2]. A CSN program (figure 1) is composed of 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.

A sensor $\vec{P} :: M^{p,r}_{e}$ 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 ($p$), transmission power ($r$), 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 func-
tion \( l \) (with arguments \( \vec{v} \)) in some value \( v \). Value \( v \) may be an anonymous module, the top level module, if the target is \texttt{loc}, or the broadcast address, if the target is \texttt{net}. In the last case, the call is broadcast to the network neighborhood of the sensor. Installing or replacing functions in a module can be done with the construct \( v.\texttt{install} \ v' \), 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 \texttt{let} construct allows processes to create local variables to hold intermediate values in computations and the construction of arbitrarily complex data structures. A sequential composition construct can be easily obtained as syntactic sugar from the \texttt{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.

The semantics of the calculus is presented following the standard practice in process calculi: a reduction relation coupled with a structural congruence relation. Together they define the rules by which CSN programs may be executed. The details of these relations can be found in [2].

**Small Programming Examples.** We sketch a simple ping program. We denote as \texttt{Ping} and \texttt{Sink} the modules installed in any of the anonymous sensors in the network and in the sink, respectively. Each sensor has a ping function that when called broadcasts a forward call to the network with its MAC address \( m \), and broadcasts another ping call to propagate the call in the network. The sink has a distinct implementation of the forward function. Any incoming call logs the MAC address given as argument. So, the overall result of the call \texttt{net.ping()} in the sink is that all reachable sensors in the network will, in principle, receive this call and will flood the network with their MAC addresses. These values eventually reach the sink and get logged.

\[
\begin{align*}
\texttt{Ping} & = \{ \texttt{ping} = () \texttt{let} \ m = \texttt{loc.macro}() \texttt{in} \texttt{net.forward}(m); \\
& \quad \texttt{net.ping}(); \\
\texttt{forward} = (x) \texttt{net.forward}(x) \} \\
\texttt{Sink} & = \{ \texttt{forward} = (x) \texttt{log.mac}(x) \} \\
\llbracket\texttt{net.ping()}, \texttt{Sink}\rrbracket & | \llbracket\{\}, \texttt{Ping}\rrbracket | \ldots | \llbracket\{\}, \texttt{Ping}\rrbracket
\end{align*}
\]

The above example assumes we have some means of bootstrapping the network, namely, of deploying the code to the sensor nodes. At boot time, we call a deploy function, 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.

\[
\text{Ping} = \{ \text{ping} = () \mapsto \text{let } m = \text{loc.mac_addr()} \text{ in net.forward}(m);
\]

\[
\text{net.ping()}
\]

\[
\text{forward} = (x) \mapsto \text{net.forward}(x) \}
\]

\[
\text{Boot} = \{ \text{deploy} = (x) \mapsto \text{install } x; \text{net.deploy}(x) \}
\]

\[
\text{Sink} = \{ \text{forward} = (x) \mapsto \text{log_mac}(x) \}
\]

\[
[\text{net.deploy[Ping; net.ping()], Sink}] \mid [\{\}, \text{Boot}]\ldots|\{\}, \text{Boot}
\]

**Current and Future Work.** In [2] we have further introduced a static type system for the calculus that enables safe programming of sensor networks. This allows typed sensor network programs to be checked at compile time, allowing the premature detection of programs that would produce run-time errors. As a first step towards establishing the *type safeness* of the model, we obtained a subject reduction result for the type-system and are currently working on proving that well typed programs do not generate run-time protocol errors. We are also working on proving the correctness of programs written in our system by using the notion of behavioral equivalence and the associated tools from process calculi theory.

Finally, we implemented an interpreter for the calculus that we are using to develop programs and to test the model robustness and expressiveness. Our next step will be to directly support the programming of hardware sensor networks by developing a compiler and a run-time system that interfaces with one of the available OSes for sensing devices.

**References**

