Sensor Network Design: Introducing Active Sensor Process

Jin Song Dong[1], Jun Sun[1] ⋆, Xian Zhang[1], and Jing Sun[2]

[1] School of Computing, National University of Singapore
{dongjs,sunj,zhangxi5}@comp.nus.edu.sg
[2] Department of Computer Science
The University of Auckland
j.sun@cs.auckland.ac.nz

Abstract. Wireless sensor network has shown a wide range of promising applications as well as challenges. In recent years, sensors have been equipped with significant processing, memory, and communication capabilities. Due to its distinguishable features, software and system design for sensor network becomes a complicated task. We believe that software engineering community can contribute to the development of sensor network systems by providing new modelling and design techniques. This paper proposes a process-based language (named *Active Sensor Process*) for high-level sensor networks specification and design. The language is introduced through various examples which represent a wide range of behavior patterns for sensor networks.

1 Introduction

In recent years, wireless sensor networks have showed a wide range of promising applications in a variety of domains, e.g., environmental monitoring, acoustic detection, smart spaces, inventory tracking, etc. With the development of sensor technology and electronic miniaturization, sensor integration makes it possible to produce extremely inexpensive sensing devices. The sensors have been equipped with significant processing, memory, and wireless communication capabilities. Thus, they are capable of performing complex in-network computation besides sensing and communication. The unique features of wireless sensor network present unique challenges for network modelling, simulation, verification, synthesis, etc. In the work, the following are identified as the rather unique characteristics of wireless sensor networks [1].

- In general, we categorize sensor networks as distributed reactive hybrid systems. Sensors may have continuous interfaces for sensing/actuating as well as discrete message input/output for inter-sensor communication.
- Sensor network nodes are data-centric. Sensor network applications are unlikely interested in the state of an individual sensor. Rather, applications

⋆ Dr. Sun is the corresponding author. Phone: +65 65162834. Fax: +67794580

focus on the data generated by sensors. There may not be a unique global address (like IP in the Internet) associated with each sensor. The data generated at a sensor node is named by attributes and applications request data matching certain attribute values. Date-centric routing is favored over traditional end-to-end routing.
- Sensor networks must be remotely reconfigurable. In traditional networks like the Internet, the nodes (personal computers) are maintained by human beings. Sensor networks however may not be accessible after employment. They may be embedded in physical structures, or thrown into inhospitable terrain. For such system to be effective, we must be able to remotely reconfigure the nodes in order to cope with refined or even completely new tasks.
- Sensor networks are application-aware. Traditional networks are designed to accommodate a wide variety of applications. Sensor networks are however application-specific, i.e., they are configured and deployed for a specific application. Thus, they are designed with the knowledge of the types of sensors, the geography, the data types generated by the sensors, etc.

In recent years, a number of sensor devices and sensor programming language have been actively developing. However, there are few sensor specification and design language proposed. A formal system design requires construction of a high-level description or a mathematical model of the system, which can later be used for a variety of system analysis tasks such as simulation, verification, performance evaluation and synthesis. Performed early in the design stage, such modelling and analysis offers the promise of a systematic approach. In particular, an executable system design provides a first-hand experience of fault detection as well as system measures. Due to the distinguishable features, system specification and design for sensor networks is not a trivial task. More importantly, we discover that existing specification languages are not effective at modelling sensor networks. For instance, the classic CSP [2] and its extension rely on lock-step synchronization between processes, which is overwhelming for inter-sensor communication. Moreover, CSP processes are not dynamically reconfigurable. The π-calculus [3], on the other hand, supports network dynamic reconfiguration by changing the channel names, which we think is in-effective for modelling data-centric sensor network routing. Plus, both process algebras lack the support of continuous sensing and actuating from and to the environment.

In this paper, we propose a specification language for sensor networks namely *Active Sensor Process* (shorted as Asp, *active* as in Active Message [4]). It refines the CSP and π-calculus to cope with the unique features of sensor networks. For instance, unique language constructs are developed to allow data-centric routing as well as sensor reconfiguration. The focus is the dynamic and interactive behaviors of sensor network nodes. Therefore, our language is event-based. However, because sensor networks may perform complex in-network computation, we also support data variables and functions. Our aim is to support systematic specification or design of wireless sensor networks at a relatively high level. The challenge is that the modelling should not be too general to miss relevant details of the sensors, but also not too complicated in order to efficiently solve interesting system analysis problems.

The research on sensor networks has been influenced by both the traditional network community and the database community. The former group tends to focus on finding efficient communicating protocol, e.g., [4–6]. The latter focuses on in-network data aggregation and data querying, e.g., [7–9]. Other works include building sensor systems and their middleware support [10–12]. There have been proposals for domain specific languages which present programming models for writing sensor network programs [13, 9, 14, 15]. We believe software engineering community can contribute to the development of sensor network systems by providing new modelling and design techniques capturing high level system behaviors. To our best knowledge, Asp is the first design language proposed for generic sensor network modelling. The previous closest is the notion of SensorML [16], which offers a modelling language for sensor processes. Comparing to SensorML, our language models the dynamic behaviors of sensor networks. Because Asp roots in traditional formal methods, it offers a notion with formal semantics, which we believe is essential for sensor network analysis such as simulation, verification and synthesis. In the formal method community, there are languages proposed for modelling hybrid or mobile reactive systems, e.g., Ambients [17], KLAIM [18], TCOZ [19], various extensions to pi-calculus [20], automata [21, 22], etc. The synchronous communication makes these language improper for sensor modelling. Moreover, because of the unique features of sensors, applying existing formalisms may not be possible or optimal.

The remainder of the paper is organized as follows. Section 2 explains the constructs of Asp using illustrative examples. Section 4 is devoted to specification of typical sensor network applications. Section 3 discusses the operational semantics of Asp. Section 5 concludes the paper with possible future works.

2 Active Sensor Process

In general, an Asp process is of the form $P(x_1, \cdots, x_n, X_1, \cdots, X_m)$ where P is the process name and x_i and X_j are parameters of the process. The parameters are partitioned into two groups. $x_1, \cdots, x_n$ are a finite number of typed data variables. A data variable x_i may model a part of the *context* or a local data store, e.g., the reserved variable name **loc** represents the location of the sensor and **ran** represents its radio range. $X_1, \cdots, X_m$ are a finite number of process variables. A process variable X_j captures a part of reconfigurable behaviors of the sensor. Value of both data and process variables can be changed by assignment. An assignment to a process variable changes explicitly the behaviors of the sensor node. When invoked, a process's parameters will be identified with the variables in the calling process. A process is defined by a process expression. In the following, we will use names starting with a capital letter to denote processes.

2.1 Sensing and Actuating

A sensor node may communicate with its local continuous environment via sensing and actuating. At the same time, it may communicate with other sensor nodes or data sinks via discrete messaging. A sensor node can sense data from its ever changing environment so as to detect certain phenomenon or to be aware of its context. Let s be a sensing channel. The process $s \odot x \rightarrow P(x)$ reads the value x of s from its (local) environment. If the environment changing is specified as a continuous function, then whenever a value is sensed through s at a time t, the value x equals to the value of the continuous function at that time. There may be multiple sensing channels on one sensor node. Different sensing channel may be dedicated to different phenomenons.

Example 1. A light sensor of a camera detects the light condition and then the camera decides what to do next.

$$\mathit{daylight} \odot x \rightarrow \mathit{React}(x)$$

where *daylight* is a sensing channel, x is the sensed value and *React(x)* is a process whose behaviors depend on the value of x. □

The data sensed by a sensor node may come from the external environment or another sensor node in the system. A sensor node influences its environment through actuating. A parameter of a sensor process may be declared as an actuating variable in the form $x : T\circledast$, where T is a type and $\circledast$ suggests that its value can be accessible by the environment (e.g., other sensor nodes). Semantically, an actuating variable effectively defines a global variable (within the radio range), i.e., a sensing channel with the same base name can read its value universally. An actuating variables can be hidden from part of the system by using the hiding operator (see later examples).

Example 2. Following the camera example, the camera screen shows lively the current light condition on the screen.

$$\mathit{Camera}(\mathit{illumination} : \mathbb{N}\circledast) \mathrel{\widehat=} \mathit{daylight} \odot x \rightarrow \mathit{Camera}(x)$$

where $\mathbb{N}$ is the set of natural numbers[3]. After sensing the day light level from the environment, the *illumination* is set to the outside light level and then the process terminates.

$$\mathit{Display}(\mathit{print} : \mathbb{N}) \mathrel{\widehat=} \mathit{illumination} \odot x \rightarrow \mathit{Display}(x)$$

The above defines a local display device, which gets the value of *illumination* through sensing and prints it on the screen. There is no matching variable named *daylight*, which suggests the channel *daylight* reads value from the external environment. □

[3] Simple types like Boolean, integer, string and set are supported.

2.2 Messaging

Sensor nodes which form a network may communicate with each other, e.g., through radio transmission. We refer it as messaging. There are unique characteristics about inter-sensor messaging. First of all, messaging between different nodes is almost always asynchronous. Depending on the communication media and the transfer rate, it may take considerably long time with no guarantee to be received. This is different from CSP-style lock-step synchronization [2]. Secondly, in an active sensor network, not only data can be sent, processes encapsulated with data and control states shall be able to propagate through the network. We believe that active message [4] is necessary for code propagation as well as process migration. For example, part of the sensors' programs then may be updated to cope with the changing of the task requirements. Lastly, there may not be a global identity for each sensor and thus end-to-end communication is unlikely in sensor networks. Because sensors experience a much higher failure rate (comparing to the Internet) and may be mobile, communication through channels shared by a pre-determined set of processes (as in CSP [2]) is undesirable. For instance, there could be a dense sensor deployment in a small area. For an individual sensor, there is no way to determine before hand what the processes are to communicate with. Thus, we introduce the following unique constructs to abstract inter-sensor messaging. The key idea is that sensors shall perform their tasks independently regarding to the network topology.

$$!\,[b]X \rightarrow P(x_1, \cdots, x_n, X_1, \cdots, X_m) \qquad \text{– output}$$
$$?\,[b]X \rightarrow P(x_1, \cdots, x_n, X_1, \cdots, X_m, X) \qquad \text{– input}$$

where b is a Boolean formula over the context of the *receiving* sensor node, namely a *receiving guard*. Intuitively, when a sensor broadcasts a message, it has no idea who might be out there listening on the channel. Thus, it uses the Boolean expression to specify who the intended receivers are. The guard is considered as true if omitted. X is called a *gradient* [5]. It could be an (compound) abstract event, a tuple or a complex process itself.

Example 3. Suppose the sensors have been deployed around certain phenomenon (e.g., earthquake). It is not easy to know the exact location of the sensors (e.g., they are thrown from an aircraft). We now want to update the sensors located within a short distance to the phenomenon.

$$![\mathbf{loc} - \mathit{pheloc} < 5 \vee \mathit{pheloc} - \mathbf{loc} < 5]\mathit{program.NewPro}$$

where **loc** is the receiver's position, *pheloc* is the location of the phenomenon (which is part of the sensors' knowledge). Both variables are of type integer. The communication guard says that only those sensor nodes within 5(m) to the phenomenon are the intended receivers. The *program.NewPro* is the message itself, in which *program* is a channel name (i.e., a name tag indicating what is the content of the message) and *NewPro* contains the new instructions. A channel can be implemented on a separate port or radio frequency or as simple as a flag bit in the message package. *NewPro* is a process itself (we omit its definition for simplicity). □

Once a sensor receives the message, the *receiving guard* is evaluated over the receivers' context. If it is true, the sensor is an intended receiver and the gradient received becomes part of the receiver's context. Otherwise, the message may be ignored as if it is never received or be forwarded by the sensor.

Example 4. Following Example 3, the receivers can be designed as the following:

$$?[X]program.Y \rightarrow Install(Y) \lhd X \rhd ![X]program.Y \rightarrow Skip$$

Once an input on the channel *program* is received, the sensor decides on how to react. If the guard evaluates to true that the new instructions are installed, otherwise, the message is forwarded. Alternatively, an end receiver may be designed as,

$$?program.Y \rightarrow Install(Y)$$

Notice that there is no communication guard at the receiving end. If the communication guard is evaluated to false at message arrival, the process keeps waiting. Thus, it makes no difference if a receiver does not receive the message or it does but fails the *guard*. □

In Asp, we assume that the output is broadcasted so that all sensors within the range may receive it. Yet only those intended ones will process it. A receiving sensor node may only listen on channels of its interest. Broadcasting is favored in sensor networks, but is certainly not the only way of messaging. If the sensor knows exactly which of its neighbors should be the receiver, it may set the guard as $id = x$ (where x is any locally unique neighbor identifier, e.g., MAC addresses or Blue tooth cluster address) and perform one-to-one communication. We remark that inter-sensor messaging is always asynchronous because in sensor networks processing time per bit communicated is plentiful, i.e., CPUs are fast and bandwidths are low.

Because the communication guard is to be evaluated over the receivers' context, the sending sensor shall have an external frame of reference for application-specific information such as sensor types, geography and pre-defined attributes. This is undesirable in traditional general purpose distributed systems because it may be against Dijkstra's principle of *loose coupling* [23]. However, we believe that constrained, application-specific domains such as sensor networks can live beyond the principle and profit from such a communication scheme. One important benefit is that we do not need to explicitly model the changing of the network topology. A mobile sensor node changes the network topology by changing its context. For example, if a receiving sensor in Example 4 moves away from the phenomenon (so the value of **loc** has changed), though it may still be in the radio range, it will ignore the message from the sender.

2.3 Local Computation

In recent years, sensors have been equipped with significant processing and memory besides wireless communication capabilities. For example, the TinyOS motes developed by Intel Research at Berkeley [10] have a 4-8MHz microcontroller, 4-10kB of data RAM, 6-128kB of program flash memory and a radio with application-level data transmission rates of 1-20kB/s. The motes are capable of performing complex in-network computation. Thus, we need facilities to model complex local computation of sensor nodes. A design for a sensor node may consist of a set of processes. One of them is identified as the main process which determines the behavior of the sensor after initialization. The parameters of the main process identify the data space (i.e., data variables) of the sensor node as well as the set of replaceable process components (i.e., process variables).

Example 5. The complete specification of a simple *Receiver* for Example 3 is the following,

$$\begin{array}{l} \mathit{Receiver}(\mathbf{loc} : \mathbb{N}, \mathbf{ran} : \mathbb{N}, \mathit{pheloc} : \mathbb{N}, \mathit{Routine}) \mathrel{\widehat{=}} \\ \qquad \mathit{Routine} \bigtriangleup ?\mathit{program}.X \rightarrow \mathit{Receiver}(\mathbf{loc}, \mathbf{ran}, \mathit{pheloc}, X) \end{array}$$

where $\bigtriangleup$ denotes interrupt. Process *Routine* (which is of type *Process* without stating) is a parameter of the main process, indicating that it is replaceable. The system performs the *routine* until a new program is received, then it performs the new program waiting for the next program update. □

In the following, the process constructs dedicated to specification the local computation are introduced. A number of process constructs introduced by CSP [2] and Timed CSP [24] are reused (some of which have different semantics). Part of the syntax is shown in Figure 1 (refer to Appendix A for a full list). The process *Skip* terminates successfully. The process *Stop* deadlocks. A sensor node which behaves as *Stop* disappears from the sensor network since it can not communicate with the rest of the system any more. A sensor node may reach the state of behaving as *Stop* if the battery runs out or the antenna is broken. The process $\mathit{Wait}(d)$, where d is a positive real number, idles for d time units. A sensor node executing $\mathit{Wait}(d)$ may go to sleep mode or power off (to save energy). In contrast to sensors which behave as *Stop*, the sensor will wake up after exactly d time units or responding to some interrupts. A process may have local-scoped variable, which is introduced by *let*. The values of the variables may be modified by assignment.

Example 6. The following process senses certain data every once a while and then computes the average so far.

$$\begin{array}{l} \mathit{Accumulate}(n : \mathbb{N}, \mathit{average} : \mathbb{R}) \mathrel{\widehat{=}} \mathit{data} \odot x \rightarrow \mathit{Idle}(2); \\ \qquad \mathit{Accumulate}(n + 1, (\mathit{average} * n + x)/(n + 1)) \end{array}$$

where $\mathbb{R}$ is the set of real numbers and the ; denotes sequential composition. The process $P_1;\ P_2$ behaves as P_1 until it terminates and then behaves as P_2.

$$
\begin{array}{lll}
P(\cdots) ::= & \mathit{Stop} \mid \mathit{Skip} & \text{– deadlock and termination} \\
\mid & \mathit{Idle}(d) & \text{– idle for } d \text{ time units} \\
\mid & \mathit{let}\ x \bullet P(\cdots) & \text{– local variable} \\
\mid & x := v & \text{– assignment} \\
\mid & e \rightarrow P & \text{– event prefixing} \\
\mid & e \odot x \rightarrow P \ \mid \ ![b]e \rightarrow P \ \mid \ ?[b]e \rightarrow P & \text{– sensing, output and input} \\
\mid & P_1(\cdots);\ P_2(\cdots) & \text{– sequential composition} \\
\mid & P_1(\cdots) \sqcap P_2(\cdots) \mid P_1(\cdots) \Box P_2(\cdots) & \text{– choice} \\
\mid & P_1(\cdots) \langle\!| \ b \ |\!\rangle P_2(\cdots) & \text{– conditional branch} \\
\mid & P_1(\cdots) \bigtriangleup e \rightarrow P_2(\cdots) & \text{– interrupt} \\
\mid & \mu X \bullet P(X) & \text{– recursion} \\
\mid & P_1(\cdots) \parallel P_2(\cdots) & \text{– local concurrency} \\
\mid & P_1(\cdots) \mid\mid\mid P_2(\cdots) & \text{– remote concurrency}
\end{array}
$$

Fig. 1. Asp Process Syntax

The sensor node gets the value of *data* through sensing. It then computes the new average. After idling for 2 time units, process *Accumulate* (with updated parameters) is invoked. □

The process $e \rightarrow P$ is called event-prefixing. It engages e and then behaves as P, where e is a synchronization event. If the event is shared by multiple processes running in parallel, the processes must synchronize the execution of the event. A compound event can be written in the form $e_1.e_2.e_3$ where e_i is a name.

Example 7. As inter-sensor messaging takes considerably long time, local computation shall always carry on without waiting for its completion. The messaging interface of a sensor node may be designed as the following,

$$\mathit{Interface}(\mathit{average} : \mathbb{R}) \mathrel{\widehat{=}} \mathit{send} \rightarrow !\mathit{average} \rightarrow \mathit{done} \rightarrow \mathit{Interface}(\mathit{average})$$

where synchronization events are used to signal the start and the end of the message output. Event *send* is engaged right before the message (i.e., sending the average $!\mathit{average}$). After the message has been sent, event *done* is engaged.

Diversity of behaviors are specified using choices. The internal choice, $P_1 \sqcap P_2$, behaves as P_1 or P_2 nondeterministically. Nondeterminism may seem undesirable from a designer's point of view. It offers a way of abstracting irrelevant internal behaviors. The external choice, $P_1 \Box P_2$, is determined by the environment. The branches are often guarded with event-prefixing or Boolean conditions. A conditional branch is written as $P_1 \langle\!| \ b \ |\!\rangle P_2$ (as in [25]), where b is a Boolean expression over the sensor's context. It behaves as specified by P_1 if b evaluates to true otherwise P_2. A special kind of choice is interrupt. The process $P_1 \bigtriangleup P_2$ behaves as P_1 until the moment the first event of P_2 is engaged and then behaves as P_2. Recursion is defined as Tarski's weakest fixpoint as in CSP [2].

Example 8. Following Example 6 and Example 7, the local computation of the sensor node may be designed as the following,

$$\begin{aligned}&\mathit{LocalComp}(n:\mathbb{N}, \mathit{average}:\mathbb{R}) \mathrel{\widehat{=}} \mathit{send} \rightarrow (\mu\,\mathit{Accu} \bullet \mathit{data} \odot x \rightarrow \mathit{Idle}(2);\\ &\quad n := n+1;\ \mathit{average} := (\mathit{average} * n + x)/(n+1);\ \mathit{Accu})\\ &\qquad \triangle\ \mathit{done} \rightarrow \mathit{LocalComp}(n, \mathit{average})\end{aligned}$$

The events *send* and *done* are synchronized by the process *LocalComp* and *Interface*. The process *LocalComp* continues sensing data and computing average while the communication interface is busy sending. Whenever the sending is *done*, the local process is interrupted so that the newly updated data can be sent.

2.4 Concurrency

There are two kinds of concurrency in sensor networks. Firstly, different sensor nodes run independently with each other. The communication between them is through asynchronous message passing[4]. Secondly, local processes of a sensor node or computation devices connected by wire where communication delay is ignored can execute concurrently with possible barrier synchronization. We use different notations to distinguish them. The parallel composition of local processes located at the same sensor node is denoted as $P_1 \parallel P_2$ where common events of P_1 and P_2 are synchronized.

Example 9. Following the above examples, the process *Interface* and *LocalComp* is composed as,

$$\begin{aligned}&\mathit{SensorNode}(n:\mathbb{N}, \mathit{average}:\mathbb{R}) \mathrel{\widehat{=}}\\ &\quad (\mu\,\mathit{Inter} \bullet \mathit{send} \rightarrow !\mathit{average} \rightarrow \mathit{done} \rightarrow \mathit{Inter}) \parallel\\ &\quad (\mu\,\mathit{Local} \bullet \mathit{send} \rightarrow (\mu\,\mathit{Accu} \bullet \mathit{data} \odot x \rightarrow \mathit{Idle}(2);\\ &\quad n := n+1;\ \mathit{average} := (\mathit{average} * n + x)/(n+1)) \triangle\ \mathit{done} \rightarrow \mathit{Local})\end{aligned}$$

The sensor consists of two components running in parallel. The recursion *Inter* is for broadcasting the data. *Local* is for local computation.

Processes modelling different sensors can be composed using the $|||$ operator. It reads as *interleaving* and suggests that there is no inter-sensor synchronization. Both parallel composition operators are symmetric and associative. An indexed parallel composition is denoted as $|||_i\ P(i)$ or $\parallel_i\ P(i)$.

Example 10. The sensors specified by *Sender* and *Receiver* (Example 5) can be composed concurrently as the following,

$$\begin{aligned}&\mathit{Sender} \mathrel{\widehat{=}} ![\mathbf{loc} - \mathit{pheloc} < 5 \vee \mathit{pheloc} - \mathbf{loc} < 5]\mathit{pro}.\mathit{NewPro}\\ &\mathit{NewPro} \mathrel{\widehat{=}} \mu\, X \bullet \mathit{data} \odot x \rightarrow \mathit{Wait}(5);\ X\\ &\mathit{Network}(\mathit{phe\_loc}:\mathbb{N}) \mathrel{\widehat{=}} \mathit{Sender}\ |||\ \mathit{Receiver}(2, \mathit{phe\_loc}, \mathit{Stop})\\ &\quad |||\ \mathit{Receiver}(7, \mathit{phe\_loc}, \mathit{Stop})\end{aligned}$$

[4] Nearby sensors may interact through sensing/actuating.

The process *NewPro* is designed to sense data every once a while, whereas the original design is to do nothing. There are 1 sending sensors and 2 receiving sensor nodes at different locations (specified as $Receiver(2, phe\_loc, Stop) \parallel\mid Receiver(7, phe\_loc, Stop)$). Note that $P \parallel\mid P$ is not equivalent to $\mathcal{S}$.

As in CSP, complex process constructs can be composed from the simple ones. We offer a number of syntactic sugars to assist intuitive system design. Some of them are briefly introduced in the following (refer to [26] for the complete syntax of Asp). A *state guard* is written as $[b] \bullet P$, in which b is a Boolean formula. The process P is executed only if b evaluates to true.

$$[b] \bullet P \mathrel{\widehat{=}} P \langle\!| \; b \; |\!\rangle Stop$$

A process that times out after some time units is written as $P_1 \bowtie_d P_2$. The process P_2 takes control from the process P_1 if P_1 has not made a move after d time units elapsed.

$$P_1 \bowtie_d P_2 \mathrel{\widehat{=}} P_1 \,\Box\, (Idle(d);\; P_2)$$

In a *timed interrupt* $P_1 \triangle_d P_2$, the process P_2 will take control from the process P_1 if d time units have elapsed and P_1 is still running. If P_1 terminates before d time units, then the whole process terminates.

$$P_1 \triangle_d P_2 \mathrel{\widehat{=}} P_1 \triangle (Idle(d);\; P_2)$$

In sensor network design, we often need to repeat certain operations for a number of times or for some time. The process P^n repeats for n times. It is the same as the sequential composition of the n copies of P. An unbounded iteration is denoted as P^*. The process P^{*d} repeats P for exactly d time units. Lastly, periodic behaviors are common in sensor networks, e.g., every 10 second, the sensor attached with the air conditioner should sense the room temperature. The process $P^{\hookleftarrow d}$ repeats every d time units. If the process P terminates before d time units, it idles until the next round, otherwise, it is interrupted to run from the beginning.

$$P^{n+1} \mathrel{\widehat{=}} P;\; P^n \text{ and } P^0 \mathrel{\widehat{=}} Skip$$
$$P^* \mathrel{\widehat{=}} P;\; P^*$$
$$P^{*d} \mathrel{\widehat{=}} P^* \triangle_d Skip$$
$$P^{\hookleftarrow d} \mathrel{\widehat{=}} ((P \triangle_d Skip) \parallel Idle(d))^*$$

3 Executing Sensor Networks

In this section, we discuss how to simulate our sensor networks models. The operational semantics of Asp processes is presented as a set of firing rules. We then describe a simulation technique which handles sensor networks with large number of similar or identical sensor nodes.

The operational semantics of Asp processes can be defined as a set of firing rules. At the top level, an Asp process is of the form $P_1 \mid\mid\mid \cdots \mid\mid\mid P_n$ where P_i is the process specifying the behaviors of a sensor node. We first explain how each sensor node behaves and then the composed behaviors. Given a process P_i, a firing rule is of the form $P_i, \mathcal{V}_i \stackrel{x}{\hookrightarrow} P_i', \mathcal{V}_i'$ where $\mathcal{V}_i$ denotes valuations of the data and process variables, which is a set of pairs of variables and their current values. A process expression may evolve to another form either by engaging an event or by time elapsing where x is a positive real number. Given a Boolean formula b, we write $V_i \vDash b$ if and only if b is implied true in the context of the given context. $V_i \nvDash b$ if b or $\forall X \bullet b$ where X are all free variables in b evaluates to false. Intuitively, it means that a free variable could take any value of its type. The firing rules (for processes specifying the behaviors of a sensor node) are summarized in Appendix B. Notice that some of them are inspired by those presented in [27]. Let e be a local synchronization or τ or a message output.

$$\frac{P_i, \mathcal{V}_i \stackrel{e}{\hookrightarrow} P_i', \mathcal{V}_i'}{P_1 \mid\mid\mid \cdots \mid\mid\mid P_i \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \stackrel{e}{\hookrightarrow} P_1 \mid\mid\mid \cdots \mid\mid\mid P_i' \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \oplus \mathcal{V}_i'} [\ \mathit{Inter}1\ ]$$

$$\frac{P_i, \mathcal{V}_i \stackrel{?[X]Y}{\hookrightarrow} P_i', \mathcal{V}_i' \wedge P_j', \mathcal{V}_j' \stackrel{![M]N}{\hookrightarrow} P_j, \mathcal{V}_j \wedge \mathit{distance}(P_i, P_j) < P_j.\mathbf{ran}}{\begin{array}{l} P_1 \mid\mid\mid \cdots \mid\mid\mid P_i \mid\mid\mid P_j \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \stackrel{?[X]Y}{\hookrightarrow} \\ \quad P_1 \mid\mid\mid \cdots \mid\mid\mid P_i' \mid\mid\mid P_j \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \oplus \mathcal{V}_i' \oplus \{X \mapsto M, Y \mapsto N\} \end{array}} [\ \mathit{Inter}2\ ]$$

$$\frac{P_i, \mathcal{V}_i \stackrel{?Y}{\hookrightarrow} P_i', \mathcal{V}_i' \wedge P_j', \mathcal{V}_j' \stackrel{![M]N}{\hookrightarrow} P_j, \mathcal{V}_j \wedge \mathit{distance}(P_i, P_j) < P_j.\mathbf{ran} \wedge V_i \vDash M}{\begin{array}{l} P_1 \mid\mid\mid \cdots \mid\mid\mid P_i \mid\mid\mid P_j \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \stackrel{?Y}{\hookrightarrow} \\ \quad P_1 \mid\mid\mid \cdots \mid\mid\mid P_i' \mid\mid\mid P_j \mid\mid\mid \cdots \mid\mid\mid P_n, \mathcal{V} \oplus \mathcal{V}_i' \oplus \{Y \mapsto N\} \end{array}} [\ \mathit{Inter}3\ ]$$

Fig. 2. Inter-sensor messaging rules

The rule *Inter*1 presented in Figure 2 says that a local event or output may fire as long as it can be performed by the sensor node. If e is of the form $x \odot y$ (sensing), then either there is a sensor P_j such that x is an actuator of P_j and $P_j.\mathbf{ran} > \mathit{distance}(P_j, P_i)$ or x is sensing channel across the system boundary. As we mentioned previously, inter-sensor messaging uses a broadcast protocol, i.e., each message output is sent to every other sensors within the range along with a communication guard. When a sensor receives a broadcast message, it evaluates the communication guard based on its internal states v. If the guard evaluates to true, then it is possible to use the received information; otherwise the message is ignored. If certain variable in the guard is beyond its knowledge, the receiver always assume the variable could be of any value. However, the sensor may still be an intended receiver. For example, the guard is $distance < 3 \vee temperature > 25$ and *distance* is a parameter of the receiver and is of value greater than 3 but *temperature* is not its parameter.

We remark that even though message output do not need cooperation of the environment, time may still elapse before a message is output because message output in sensor networks are always not instant. For a specific application, the exact time of sending a message need to be specified if relevant, e.g., set as a parameter to the simulator. The initialization of a sensor application also needs to identify the network topology, i.e., the neighboring relationship between sensor nodes.

Simulation enables examination of possible dynamic interactions between sensors during early design stage and thus provides an inexpensive way to fault detection as well as system measurements like energy efficiency, space coverage, etc. Because how the sensor network will work depends on not only the sensors but also the environment, an additional set of parameters are required for simulation, for example, radio range, average time taken for message sending, message lost rate, the environment changing rate (specified by some continues functions), etc. For simplicity, we currently assume a reliable link layer protocol.

One unique challenge of simulating sensor networks is that there may be thousands of sensor nodes with similar or identical behavior deployed for one application. It is expensive to maintain the state information of individual sensors (because of state space explosion). In the future, we will explore a *symbolic* simulation technique where sensors are dynamically grouped and de-grouped based their data and control states (partially inspired by the work in [28]).

Given a sensor process, a behavioral partition is a tuple $(expr, v)$ where *expr* is a control state or equivalently an Asp process expression, v is the valuation of the data variables and the process variables. Let $\mathcal{P}$ be the set of behavioral partitions. Two sensors belong to the same behavioral partition if and only if they reside in the same control state and have the same valuation of the variables. Because sensor networks are data-centric and may not have global identify, sensors belonging to the same behavioral partition are indistinguishable from users' point of view. What matters is the number of sensors belong to a partition. The maximum number of partitions depends on the number of control states and the domain of the variables[5].

An abstract configuration of a network is defined as $\{(bp, n)\}$ where $bp \in \mathcal{P}$ is a behavioral partition and n is the number of sensors belonging to the partition. Let $\mathcal{C}$ be the set of abstract configuration. To simulate the sensor network, we only need to keep track of the current configuration of the network. An abstract simulation of a given sensor network is a state machine $(S, s_0, \hookrightarrow)$ where S, the

[5] We may replace v with a Boolean formula so that all valuation of the data variables satisfying the same formula are considered same. In that case, some theorem proving facility is a must.

set of states, are the set of reachable abstract configurations and s_0 is the initial configuration and $\hookrightarrow \subseteq S \times \Sigma \times S$ is the least transition relation satisfying the following: let $c_i = (expr_i, v_i)$,

$$\begin{aligned}\forall\, s_1 : \mathcal{C} \bullet ((c_1, n) \in s_1 \wedge (c_2, m) \in s_1 \wedge (expr_1, v_1) \stackrel{a}{\hookrightarrow} (expr_2, v_2)) \\ \Rightarrow s_1 \stackrel{a}{\hookrightarrow} (s_1 \setminus \{(c_1, n), (c_2, m)\} \cup \{(c_1, n-1), (c_2, m+1)\})\end{aligned}$$

where a is either an event or time elapsing. We remark that the behavioral partitions shall be created on-the-fly, i.e., only those with one or more sensors are maintained.

4 Case Studies

This section demonstrates a variety of example applications of our design language which represent a wide range of behavior patterns for sensor networks. The first example shows a complete design of a simple system, in which sensors are located in a confined area. The second example models a nontrivial code propagation and maintenance algorithm proposed for large sensor networks.

4.1 Smart Space System

In this scenario, a room contains a motion sensor, an air conditioner and a group of lights. Each light is equipped with a sensor. The motion sensor detects movement in the room and communicates with the lights. The light tunes the light level according to the daylight if there are people in the room. Otherwise, the light is turned off (tuned to 0) to save energy. The air conditioner functions on its own. Once turned on, it monitors the current room temperature so as to either start cooling or simply idle.

$$MotionSensor \mathrel{\widehat{=}} (motion \odot x \rightarrow ![type = \text{“}lightsensor\text{”}]motion.x \rightarrow Skip)^{\hookleftarrow 2}$$

The motion sensor senses movement in the room and communicates with the light sensors every 2 time units.

$$\begin{aligned}&Light(type : \mathbb{S}, illumination : \mathbb{N}) \mathrel{\widehat{=}} (?motion.0 \rightarrow \\ &\quad ?motion.0 \rightarrow ?motion.0 \rightarrow Light(type, 0)) \\ &\quad \triangle\ (?motion.1 \rightarrow lightlevel \odot x \rightarrow Light(type, 100 - x))\end{aligned}$$

where $\mathbb{S}$ denotes the set of all strings. Whenever the light receives a message from the motion sensor saying that there is movement in the room, it senses the outside light level and tunes its illumination. If it receives three consecutive messages saying there is no movement, it turns off the light.

$$\begin{aligned}&Aircon(k : \mathbb{N}, display : \mathbb{N}\circledast) \mathrel{\widehat{=}} turnon \rightarrow ((temperature \odot x \rightarrow display := x; \\ &\qquad ([x > k] \bullet start \rightarrow Skip \,\Box\, [x \leq k] \bullet end \rightarrow Skip))^{\hookleftarrow 2} \\ &\quad \triangle\ turnoff \rightarrow Aircon(k, display))\end{aligned}$$

The air conditioner can be manually turned on or off (through event *turnon* or *turnoff*). Once turned on, it will sense the room temperature and decide whether to start cooling or not. Notice that the outputs from the motion sensor is ignored by the air conditioner because it has no variable named *type*. Lastly,

$$\begin{aligned}
&\mathit{Room} \mathrel{\widehat{=}} \mathit{Aircon}(22, 22) \mathbin{|||} \mathit{MotionSensor} \mathbin{|||} \\
&\quad \mathit{Light}(\text{``}\mathit{lightsensor}\text{''}, 0) \mathbin{|||} \mathit{Light}(\text{``}\mathit{lightsensor}\text{''}, 0) \\
&\quad \mathbin{|||} \mathit{Light}(\text{``}\mathit{lightsensor}\text{''}, 0) \mathbin{|||} \mathit{Light}(\text{``}\mathit{lightsensor}\text{''}, 0)
\end{aligned}$$

4.2 Trickle Algorithm

The *Trickle* algorithm [6] is a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. A program update is propagated through the network using a "polite gossip" policy, where a sensor node periodically communicates with its neighbors to stay updated. The modelling demonstrates that Asp covers both the application and the network layers.

$$\begin{aligned}
&\mathit{Mote}(\mathit{version}, c, k, t, \tau : \mathbb{N}, \mathit{Routine}) \mathrel{\widehat{=}} \\
&\quad (\mathit{Routine} \bigtriangleup \mathit{update} \rightarrow \mathit{Skip}) \parallel \mathit{Maintain}
\end{aligned}$$

Informally, each sensor node (called *mote* in [10]) maintains a counter c, a threshold k, a timer t and a constant τ. Variable *version* is an incrementing counter which tells the current version of the *Routine*. Each mote consists of two components running in parallel, one performing regular routine (specified as *Routine*) and the other maintains the routine program (specified as *Maintain*). Notice that *Maintain* is an alias rather than a process, e.g., it has no parameters and yet it can assess all variable of the process. Alias are introduced solely for the purpose of readability, especially for complex systems. The sensor node continues running according to *Routine* until the *Maintain* component signals that *Routine* has to be updated. Then, the *Routine* is interrupted and the system restarts.

$$\begin{aligned}
&\mathit{Maintain} \mathrel{\widehat{=}} \mathit{Update} \parallel (\mathit{Gossip} \bigtriangleup \mathit{update} \rightarrow \mathit{Skip}) \\
&\mathit{Update} \mathrel{\widehat{=}}\ ?\mathit{program}.(x, \mathit{NewR}) \rightarrow (\mathit{update} \rightarrow \mathit{Mote}(\mathit{version}, c, k, t, \tau, \mathit{NewR}) \\
&\qquad\qquad \langle\!| \; x > \mathit{version} \; |\!\rangle \, \mathit{Update})
\end{aligned}$$

The *Maintain* may perform either an update or "polite gossip" so as to find out if there is a newer version of the program or there is some neighbor who is out of date. In the process *Update*, the system waits for an input for update. The first part of the input is the version of the attached program. The second is the program itself. If the program is newer, i.e., $\mathit{version} > x$, the mote is updated with the new program, otherwise, the process repeats.

$$
\begin{aligned}
&\mathit{Gossip} \mathrel{\widehat{=}} (\mathit{Talk} \parallel \mathit{Reply}) \triangleright\{\tau\}\ c := 0;\ \mathit{Gossip}\\
&\mathit{Talk} \mathrel{\widehat{=}} \mathit{Idle}(t);\ \mathit{Skip} \langle\!|\ c \geq k\ |\!\rangle !\mathit{meta}.\mathit{version} \rightarrow \mathit{Talk}\\
&\mathit{Reply} \mathrel{\widehat{=}} ?\mathit{meta}.x \rightarrow ((c := c+1;\ \mathit{Reply})\\
&\qquad \langle\!|\ x = \mathit{version}\ |\!\rangle\\
&\qquad (![\mathit{version} = x]\mathit{meta}.\mathit{version} \rightarrow \mathit{Reply}\\
&\qquad\qquad \langle\!|\ x > \mathit{version}\ |\!\rangle\\
&\qquad ![\mathit{vertion} = x]\mathit{program}.(\mathit{version}, \mathit{Routine}) \rightarrow \mathit{Reply}))
\end{aligned}
$$

The *Gossip* process captures the essence of the *Trickle* algorithm. During maintenance, the mote announces its metadata once a while (specified by *Talk*) or it listens to other motes and reacts (specified *Reply*). Every τ time units, the process restarts (and resets the counter c). In the process *Talk*, after t time units, the mote checks if it has talked too much ($c \geq k$). If it has, the mote will keep silent until the next gossip, else, it will broadcast its version on channel *metadata* and wait for its next turn to talk. In the process *Reply*, when a mote hears metadata identical to its own ($x = \mathit{version}$), it increments c. Once its value changes, the process *Mote* and all its sub-processes will execute depending on the new value. If the mote hears a new metadata ($x > \mathit{version}$), it broadcasts its own metadata to the motes with the updated program ($\mathit{version} = x$), which will trigger the receiver to send the updated program. If the mote hears an old metadata, it broadcasts the code necessary to update the mote sending the old metadata. Only the motes with the old program will receive the message.

$$\mathit{Network} \mathrel{\widehat{=}} |||_i\ \mathit{Mote}(1, 0, 2, 2, 5, \mathit{Routine})$$

This sensor network is made up of n motes with the proper initial settings. We skip the irrelevant details of *Routine* and the node that regularly puts updates into the network.

5 Conclusion

The contribution of the paper is threefold. Firstly, we proposed a high-level design language specifically for the wireless sensor network systems. Unique language constructs have been defined to cope with the unique characteristics of such systems, e.g., sensing and actuating, inter-sensor messaging, etc. Secondly, we modelled a set of common sensor network applications using the proposed Asp language. From the examples, Asp showed its high expressiveness and conciseness in formally specifying the communications and behaviors of sensor network systems. Lastly, we also investigated into the techniques for simulating the Asp models where numerous identical sensors coexists. In summary, the proposed Asp language not only offers a systematic way of modelling/specifying sensor networks, with the precise semantics defined, but also gives us a starting point for sensor network simulation, verification and synthesis. We are currently developing both textual and XML representation for Asp, based on which there are a series of tool development like simulator, verifier (e.g., [29]) and synthesizer (e.g., [30, 31]).

References

1. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next Century Challenges: Scalable Coordination in Sensor Networks. In: Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1999). (1999) 263–270
2. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall (1985)
3. Milner, R.: Communicating and Mobile Systems : the π-calculus. Cambridge University Press (1999)
4. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active Messages: A Mechanism for Integrated Communication and Computation. In: Proceedings of International Symposium on Computer Architecture 1992 (ISCA 1992). (1992) 256–266
5. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: a Scalable and Robust Communication Paradigm for Sensor Networks. In: Proceedings of International Conference on Mobile Computing and Networking (MOBICOM 2000). (2000) 56–67
6. Levis, P., Patel, N., Culler, D.E., Shenker, S.: Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In: Proceedings of Networked Systems Design and Implementation (NSDI 2004). (2004) 15–28
7. Ciancio, A.G., Pattem, S., Ortega, A., Krishnamachari, B.: Energy-efficient Data Representation and Routing for Wireless Sensor Networks based on a Distributed Wavelet Compression Algorithm. In: Proceedings of Information Processing in Sensor Networks (IPSN 2006). (2006) 309–316
8. Krishnamachari, B., Estrin, D., Wicker, S.B.: The Impact of Data Aggregation in Wireless Sensor Networks. In: Proceedings of International Conference on Distributed Computing Systems, Workshops (ICDCW 2002). (2002) 575–578
9. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an Acquisitional Query Processing System for Sensor Networks. ACM Transactions on Database Systems **30**(1) (2005) 122–173
10. Levis, P., Culler, D.E.: Maté: a Tiny Virtual Machine for Sensor Networks. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002). (2002) 85–95
11. Levis, P., Lee, N., Welsh, M., Culler, D.E.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: Proceedings of International Conference on Embedded Networked Sensor Systems (SenSys 2003). (2003) 126–137
12. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to Support Sensor Network Applications. IEEE Network **18**(1) (2004) 6–14
13. Boulis, A., Han, C.C., Srivastava, M.B.: Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In: Proceedings of International Conference on Mobile Systems, Applictions, and Services (MobiSys 2003). (2003)
14. Welsh, M., Mainland, G.: Programming Sensor Networks Using Abstract Regions. In: Proceedings of Networked Systems Design and Implementation (NSDI 2004). (2004) 29–42
15. Mottola, L., Picco, G.P.: Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In: Proceedings of Distributed Computing in Sensor Systems (DCOSS 2006). (2006) 150–168

16. Botts, M.: Sensor Model Language (SensorML). http://vast.nsstc.uah.edu/SensorML/ (2006)
17. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98, Springer-Verlag, Berlin Germany (1998)
18. de Nicola, R., Ferrari, G.L., Pugliese, R.: klaim: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing) (1998)
19. Mahony, B., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering **26**(2) (2000)
20. Gruhn, V., Schäfer, C.: Architecture Description for Mobile Distributed Systems Using Typed *i*-Calculus. Electronic Notes in Theoretical Computer Science **150**(1) (2006) 51–60
21. Alur, R., Dang, T., Esposito, J.M., Fierro, R.B., Hur, Y., Ivancic, F., Kumar, V., Lee, I., Mishra, P., Pappas, G.J., Sokolsky, O.: Hierarchical Hybrid Modeling of Embedded Systems. In: Proceedings of International Workshop on Embedded Software (EMSOFT 2001). (2001) 14–31
22. Alur, R.: Timed Automata. In: Proceedings of International Conference on Computer Aided Verification (CAV 1999). (1999) 8–22
23. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
24. Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., Roscoe, A.W.: Timed CSP: Theory and practice. In: Real-Time: Theory in Practice. Volume 600. (1992) 640–675
25. Hoare, C.A.R., He, J.F.: Unifying Theory of Programming. Prentice-Hall (1998)
26. Sun, J., Dong, J.S.: Active sensor process. http://www.comp.nus.edu.sg/~sunj/sensor.ps (2006)
27. Schneider, S.: An Operational Semantics for Timed CSP. Information and Computation **116**(2) (1995) 193–213
28. Goel, A., Meng, S., Roychoudhury, A., Thiagarajan, P.S.: Interacting Process Classes. In: Proceedings of International Conference on Software Engineering (ICSE 2006). (2006) 302–311
29. Braberman, V.A.: Automatic Verification of Real-Time Designs. In: Proceedings of International Conference on Software Engineering (ICSE 1999). (1999) 716–717
30. Bontà, E., Bernardo, M., Magee, J., Kramer, J.: Synthesizing Concurrency Control Components from Process Algebraic Specifications. In: Proceedings of Coordination Models and Languages (COORDINATION 2006). (2006) 28–43
31. Sun, J., Dong, J.S.: Design Synthesis from Interaction and State-Based Specifications. IEEE Transactions on Software Engineering **32**(6) (2006)

Appendix A: Asp Syntax

The syntax is described in an extended BNF with the following meta-symbols:

::=	– produces
$\mid$	– alternative
$[\,x\,]$	– optional x
$[\![x]\!]$	– zero or more x's
$[\![x]\!]_1$	– one or more x's

Nonterminals ending in *List* have productions according to:

$xList ::= x\{, x\}$

Nonterminal names are typically compound and abbreviated, each part commencing with an upper-case letter. Abbreviations are listed below.

Spec	– specification
Sep	– Separator
Proc	– Process
Exp	– Expression
Para	– Parameter
Prim	– Primitive
Pred	– Predicate

Terminal symbols are shown directly as they appear in *Active Sensor Process*. Metasymbols are larger than terminal symbols of the same shape as seen by $\big|, |, \big[[, \big],]$.

The order of presentation of productions is top down with definitions for nonterminals appearing after their last application. The plain text representation of ASP models is briefly teletyped.

Spec $::= Def \ \llbracket Sep \ Def \rrbracket$
A specification contains a set of definitions, separated by some separators, e.g., line-breaker.

Def $::= TypeDef \ \big| \ ProcDef$
Besides process definitions, we support type definitions. User-defined classes have not yet been included because we think complex hierarchical data structure is overwhelming for current sensors.

TypeDef $::= Name \ \big[\ NameList \ \big] == TypeExp$
An abbreviation type is defined in terms of existing types using a type expression. For example, $Salary == \mathbb{N}$ where $\mathbb{N}$ represents the set of natural numbers.

TypeExp $::= \mathbb{B} \ \big| \ \mathbb{N} \ \big| \ \mathbb{Z} \ \big| \ \mathbb{R} \ \big| \ \mathbb{S} \ \big| \ \mathbb{P} \ TypeExp$
$\big| \ TypeExp \llbracket \times TypeExp \rrbracket_1$
$\big| \ \{Name \ \big[\ NameList \ \big] \} \ \big| \ \{NameList : TypeExp \mid Pred\}$
The pre-defined types include Boolean, natural numbers, integers, real numbers, string and sets. Types can be composed to form tuple types. For example, the type $SalaryRecord == \mathbb{S} \times \mathbb{N}$ contains 2-tuples where the first component is a string (name) is the second is the salary number, e.g., (John, 1000). Enumerate types are defined

in form of $\{a, b, c\}$. Lastly, users can define a constrained set as a type. For example, $\{x : \mathbb{N} \mid 0 < x < 100\}$. The types can be defined in plain text using a set of reserved keywords and symbols.

ProcDef $::= \mathit{Name}(\mathit{ParaDefList}) \mathrel{\widehat{=}} \mathit{ProcExp}$
A process definition contains a name and an optional list of parameter definitions. The computation logic of the process is defined by the process expression.

ParaDef $::= \mathit{Name}[\mathit{NameList}] : \mathit{TypeExp}[\circledast]$
A parameter definition contains one or more names (sharing the same type) and a type definition. The optional symbol $\circledast$ is used to indicate actuators.

ProcExp

$$
\begin{aligned}
::= {} & \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Idle}(\mathit{Exp}) \mid \mathit{PrimExp} \\
\mid {} & \mathit{Name} \odot \mathit{Exp} \rightarrow \mathit{ProcExp} \\
\mid {} & ![\mathit{Pred}]\mathit{Exp} \rightarrow \mathit{ProcExp} \\
\mid {} & ?[\mathit{Pred}]\mathit{Exp} \rightarrow \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \sqcap \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \,\square\, \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \langle\!| \ \mathit{Pred} \ |\!\rangle \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \bigtriangleup \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \mid\mid\mid \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \parallel \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp};\ \mathit{ProcExp} \\
\mid {} & [\mathit{Pred}] \bullet \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \bowtie_{\mathit{Exp}} \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp} \bigtriangleup_{\mathit{Exp}} \mathit{ProcExp} \\
\mid {} & \mathit{ProcExp}^{*} \mid \mathit{ProcExp}^{\mathit{Exp}} \\
\mid {} & \mathit{ProcExp}^{*\mathit{Exp}} \mid \mathit{ProcExp}^{\hookleftarrow \mathit{Exp}} \\
\mid {} & \mid\mid\mid_{\mathit{Exp}} \mathit{ProcExp} \mid \parallel_{\mathit{Exp}} \mathit{ProcExp} \\
\mid {} & (\mathit{ProcExp})
\end{aligned}
$$

The process expression defines exactly the computational logic of the process. For exact meaning of all the constructors, please refer to our document on the semantics.

PrimExp $::= \mathit{Name} := \mathit{Exp} \mid \mathit{ProcRefExp}$
A primitive command is either an assignment or a process referencing.

ProcRefExp $::= \mathit{Name}\left[(\llbracket \mathit{ExpList} \rrbracket)\right][\llbracket \mathit{MapExp} \rrbracket]$
A process referencing contains an optional list of parameters and an optional list of parameter replacement.

MapExp $::= \mathit{Name} \mapsto \mathit{Name}$
For example, $\mathit{Routine}(x)[x \mapsto 5]$ is semantically equivalent to $\mathit{Routine}(5)$.

BoolExp	$::=$ *Exp*
Exp	$::=$ *Exp Infix Exp* $\mid$ *Exp.Exp* $\mid$ *Prefix Exp* $\mid$ *Exp Postfix* $\mid$ *Name* $\mid$ $\varnothing$ $\mid$ *this.Exp* $\mid$ (*Exp*)

This syntax does not define the details of the productions for *Infix*, *Prefix*, *PostFix*, *Seq*, *Pred* and *Name*.

Appendix B: Operational Semantics

Let e be either an abstract event, a data sensing, a message input/output, the $\checkmark$ or a time elasping. Let τ be an internal transition which takes place whenever possible. Notice that τ transitions are never synchronized. Let x be either an event e or τ.

$$\frac{}{\mathit{Skip}, \mathcal{V} \overset{\checkmark}{\hookrightarrow} \mathit{Stop}, \mathcal{V}}$$

$$\frac{}{e \to P, \mathcal{V} \overset{e}{\hookrightarrow} P, \mathcal{V}}$$

$$\frac{}{\mathit{Idle}(d), \mathcal{V} \overset{d_1}{\hookrightarrow} \mathit{Idle}(d - d_1), \mathcal{V}}$$

$$\frac{}{\mathit{Idle}(0), \mathcal{V} \overset{\tau}{\hookrightarrow} \mathit{Skip}, \mathcal{V}}$$

$$\frac{}{\mathit{let}\ x \bullet P, \mathcal{V} \overset{\tau}{\hookrightarrow} P, \mathcal{V} \oplus \{x \mapsto \_\}}$$

$$\frac{}{x := v, \mathcal{V} \overset{\tau}{\hookrightarrow} \mathit{Skip}, \mathcal{V} \oplus \{x \mapsto v\}}$$

$$\frac{P_1, \mathcal{V} \overset{x}{\hookrightarrow} P_1', \mathcal{V}'}{P_1;\ P_2, \mathcal{V} \overset{x}{\hookrightarrow} P_1';\ P_2, \mathcal{V}'}$$

$$\frac{P_1, \mathcal{V} \overset{\checkmark}{\hookrightarrow} P_1', \mathcal{V}}{P_1;\ P_2, \mathcal{V} \overset{\checkmark}{\hookrightarrow} P_2, \mathcal{V}}$$

$$\frac{P_1, \mathcal{V} \xhookrightarrow{x} P_1', \mathcal{V}'}{P_1 \mid P_2, \mathcal{V} \xhookrightarrow{x} P_1', \mathcal{V}'}$$

$$\frac{P_2, \mathcal{V} \xhookrightarrow{x} P_2', \mathcal{V}'}{P_1 \mid P_2, \mathcal{V} \xhookrightarrow{x} P_2', \mathcal{V}'}$$

$$\frac{}{P_1 \langle\!| \; b \; |\!\rangle P_2, \mathcal{V} \xhookrightarrow{\tau} P_1, \mathcal{V}} \; [\; V \vDash b \;]$$

$$\frac{}{P_1 \langle\!| \; b \; |\!\rangle P_2, \mathcal{V} \xhookrightarrow{\tau} P_2, \mathcal{V}} \; [\; V \nvDash b \;]$$

$$\frac{P_1, \mathcal{V} \xhookrightarrow{x} P_1', \mathcal{V}'}{P_1 \triangle P_2, \mathcal{V} \xhookrightarrow{x} P_1' \triangle P_2, \mathcal{V}'}$$

$$\frac{P_2, \mathcal{V} \xhookrightarrow{x} P_2', \mathcal{V}'}{P_1 \triangle P_2, \mathcal{V} \xhookrightarrow{x} P_2', \mathcal{V}'}$$

$$\frac{P_1, \mathcal{V}_1 \xhookrightarrow{x} P_1', \mathcal{V}' \wedge e \in \Sigma_{P_1} \setminus \Sigma_{P_2}}{P_1 \parallel P_2, \mathcal{V} \xhookrightarrow{x} P_1' \parallel P_2, \mathcal{V} \oplus \mathcal{V}_1'}$$

$$\frac{P_2, \mathcal{V}_2 \xhookrightarrow{x} P_2' \wedge e \in \Sigma_{P_2} \setminus \Sigma_{P_1}}{P_1 \parallel P_2, \mathcal{V} \xhookrightarrow{x} P_1 \parallel P_2', \mathcal{V} \oplus \mathcal{V}_2'}$$

$$\frac{P_1, \mathcal{V} \xhookrightarrow{e} P_1', \mathcal{V} \wedge P_2, \mathcal{V} \xhookrightarrow{e} P_2', \mathcal{V}}{P_1 \parallel P_2, \mathcal{V} \xhookrightarrow{e} P_1' \parallel P_2', \mathcal{V}}$$