Reconfigurable sensor network architecture for distributed measurement systems

Szilárd Jaskó  
Dept. Electrical Engineering and Information Systems  
University of Pannonia  
Veszprém, Hungary  
jasko.szilard@uni-pen.hu

Gyula Simon  
Dept. Computer Science and Systems Technology  
University of Pannonia  
Veszprém, Hungary  
simon@dcs.uni-pannon.hu

Abstract—A reconfigurable sensor-networking architecture constructed from lightweight functional elements, running on small virtual machines in parallel is proposed. The proposed architecture provides a flexible measurement system that is easily adaptable to changing system requirements in run-time. It is based on Communicating Sequential Processes and thus provides the possibility of verification and checking of system correctness.

Keywords - sensor networks; reconfigurable; services; virtual machine; CSP

I. INTRODUCTION

Sensor networks are often used as distributed measurement systems, which autonomously collect data, detect events of interest, or even provide some means of intervention in the monitored environment. Sensor networks may operate for years after deployment, so such systems are especially useful where there is limited access to the site and maintenance is difficult. A natural requirement of long-life autonomous measurement systems is the ability to adapt to changing requirements. During the lifetime of the system, the environment, the measurement/observation goals, the focus, or even the purpose of the system may change. Additionally, possible bugs may be discovered after system deployment.

In this paper, a reconfigurable sensor-networking architecture will be proposed, which can be used to construct distributed measurement systems from lightweight functional elements. The network can be efficiently reconfigured to reflect actual requirements just by adding or removing some functional elements. Another advantage of the proposed architecture is that it provides the natural possibility of systematic verification of system correctness, due to the applied syntax, which is based on Communicating Sequential Processes (CSP).

New contributions of the paper are the following:
- We present the CSP Virtual Machine architecture for sensor nodes, which enables code generation and execution directly from formal specification (CSP).
- We define CSP-based funclets, which are basic building blocks of highly-flexible reconfigurable measurement services.
- The proposed system is inherently eligible for formal verification.
- We design and implement the virtual machine on TinyOS and illustrate its effective operation.

II. RELATED WORK

Much work has been dedicated to the development of more versatile and reliable sensor networking systems. One key aspect of versatility is efficient dynamic program management, which allows applications to be updated during run-time. Several solutions have been proposed for reprogrammable or reconfigurable networks, and some of them were tested extensively in real applications.

System reconfiguration usually involves reprogramming the system. Since such systems often must be reprogrammed on-site, radio reprogramming methods are often applied. This approach is available in most systems, e.g. in TinyOS [1], where the application code is downloaded to each node in the system, using the available communication mechanisms of the sensor network. The disadvantages of this approach are that (1) the full application must be downloaded, regardless of the nature and extent of the modification. (In embedded systems the operating system must be downloaded as well.) Thus, making changes to the system can be costly. Another disadvantage of this approach is that (2) program execution must be interrupted, at least briefly. Thus, reprogramming is rarely done in order to perform major system updates. A possible solution to conventional code deployment and update problems was proposed in SOS, which is a more flexible dynamic operating system for sensor networks [2]. The kernel of SOS implements commonly used services, while the dynamic application modules can be loaded and unloaded at run-time with minimal system interruption and small overhead.

The concept of a Virtual Machine (VM) is well-known; VMs were originally developed to support platform independent execution. The best-known example is perhaps the Java VM. This idea has also been applied to sensor networks. The first VM to run on top of TinyOS was Maté, a byte code interpreter providing a platform independent development environment and an on-the-fly reprogramming method [3]. Maté then evolved to Application Specific Virtual Machine (ASVM) [4]. SensorWare defines lightweight and mobile control scripts that allow the effective utilization of computation, communication, and sensing resources of sensor nodes. In addition, new services can dynamically be installed, allowing the abstraction level best fit to the application to be used [5]. The VM* implementation is a hybrid approach, using both native and virtual (Java-like) code segments, to improve effectiveness [6].
Naturally, VMs have their disadvantages, too. For instance, they execute slower and consume more energy than their native counterparts, due to the induced extra layer of indirection. From the reconfiguration point of view, however, the main advantage of using VMs is that when an application update is required, only the concise application code (or possibly just a segment of it) must be downloaded to the nodes, instead of the full execution environment.

III. SYSTEM SPECIFICATION

A. Mathematical model (CSP)

Hoare’s language of Communicating Sequential Processes (CSP) [7, 8, 9] is a notation for describing concurrent systems and the interaction patterns between the component processes. It has a wide range of application from programming languages [10] to verification of safety protocols [11]. A CSP system is made up of independently running sequential processes, which communicate with each other by passing messages.

There are various tools for checking CSP implementations. There are animators which make it possible to write arbitrary process descriptions and to interact with them [12]. There are also refinement checkers which explore all of the states of a process [13]. CSP is very useful for debugging failures, discovering deadlock or livelock, and it is an ideal language for verification, validation and test processes [14, 15, 16].

In the proposed architecture, the system functionality is expressed with lightweight communicating processes running in parallel, and the system is described by a CSP-like notation.

B. Funclets

Each sensor is constructed from small functional elements, called funclets. A funclet can be considered a special CSP process with the following restrictions: (1) The process can start after an external triggering event (e.g. message reception, timer event), and (2) the rest of the events in the funclet do not depend on the environment (e.g. calculations, message transmissions). Thus a funclet is defined as follows:

\[ \text{funclet} ::= \text{external}_\text{events}, \text{internal}_\text{events} \]
\[ \text{external}_\text{events} ::= \text{channel}_\text{read} | \text{manage}_\text{event} \]
\[ \text{internal}_\text{events} ::= \text{internal}_\text{event} \]
\[ \text{internal}_\text{event} ::= \text{channel}_\text{write} | \text{message}_\text{operations} | \text{assignment} | \text{flow}_\text{control} \]

where external events can be channel_read (incoming messages, measurement, or timing event) and manage_event. Internal events may be channel_write (send message, start measurement, or configure timer), message_operations (message compose and decompose), assignment (arithmetic operations and handling of variables) and flow_control (if statement or loop).

Internal and external events are also described with the corresponding CSP syntax with some simplification and modification for practical purposes. Note that despite the slight notational differences a funclet has always a one-to-one correspondence to its CSP process counterpart.

In a simple example process, ECHO sends a hello message whenever it gets a request. A possible CSP solution is the following:

\[ \text{ECHO} = \text{radio?request} \rightarrow \text{radio!hello} \rightarrow \text{ECHO} \]

With the simplified funclet syntax, the process can be described as follows:

\[ \text{funclet(ECHO)} = \text{radio?request, radio!hello} \]

which contains the external event radio?request (receives a message from channel radio and stores it in variable request) and one internal event radio!hello (on channel radio sends a message, stored in variable hello).

In the next example, process ECHO1 sends only one hello message. It replies to the first received request and ignores the rest of them.

\[ \text{ECHO1} = \text{radio?request} \rightarrow \]
\[ ((\text{send}_\text{counter}.\text{assign.1} \rightarrow \text{radio!hello}) \]
\[ <\text{send}_\text{counter}=0>) \]
\[ \rightarrow \text{ECHO1} \]

The funclet equivalent which bears resemblance to traditional programming languages is the following:

\[ \text{funclet(ECHO1)} = \text{radio?request, if(\text{send}_\text{counter}=0)} \]
\[ \{\text{send}_\text{counter}.\text{assign.1, radio!hello} \}
\]

The third example illustrates loops. Process ECHO2 sends three hello messages for the first received request and one hello to the rest. The CSP representation of the process is the following:

\[ \text{ECHO2} = \text{radio?request} \rightarrow \]
\[ ((\text{send}_\text{counter} < 3) \rightarrow \text{radio!hello} \rightarrow \]
\[ \text{send}_\text{counter}.\text{assign. send}_\text{counter} + 1) <\text{send}_\text{counter}=0>) \]
\[ \text{radio!hello} \rightarrow \text{ECHO2} \]

The corresponding funclet, using the more familiar for loop and incremental operator notations, is the following:

\[ \text{funclet(ECHO2)} = \text{radio?request, if(\text{send}_\text{counter}=0)} \]
\[ \{\text{for}(\text{send}_\text{counter}<3; \text{send}_\text{counter}++)(\text{radio!hello}) \}
\[ \text{else }\{\text{radio!hello} \}
\]

The programming model includes dynamic funclet management, so any funclet or the whole program (which is just a set of funclets) can be taught, deleted, activated or deactivated on a particular sensor node or the whole sensor network.

The full syntax definition of funclets, along with the description of semantics, is included in the Appendix.
C. Process management

Sensors are constructed from funclets, using the choice operator between funclets. An important consequence of this fact is that only one funclet is active at a time; thus only one funclet can access system resources at any given time. Therefore, from an implementation point of view, funclets are atomic operations. Although funclets can access global resources in the programming model, race conditions are impossible. Naturally, external events can trigger funclets at any time (and these events are promptly registered by the system when they do occur), but the triggered funclets are executed only when the currently running funclet terminates.

The states and the possible transitions between them are shown in Fig. 1. Funclets added to a sensor are initially in Inactive state. An inactive funclet cannot be executed, no matter what events are received by the sensor. When an activation command arrives, the funclet changes to Active state, where it becomes ‘sensitive’ to certain external events. When such an event arrives, the funclet goes to Triggered state. The virtual machine eventually schedules the funclet, and the funclet is in Running state while it is being executed. Only one funclet can run at a time, and its execution cannot be interrupted by any other funclets. When the funclet’s last internal event is executed, the funclet goes back to Active state. A funclet may also be deleted.

In the virtual machine, an event queue, a funclet list and a triggered funclet list are used. Received external events are put in the FIFO event queue, and when the triggered funclet list is empty the first event in the event queue is evaluated, and the funclets triggered by this event are moved to the triggered funclet list. Note that any number of funclets can be triggered by one event. The triggered funclets are executed in sequence (with no priorities). If there is no triggered funclet, the next event in the event queue is evaluated again.

At the network level, sensor processes are running in parallel. The executions of the sensor processes are independent of each other. A sensor can communicate with other sensors via messages and can affect the behavior of other sensors in this way.

The sensor network can be reconfigured by the management command /Teach and /Set. Command /Teach distributes the funclet in the network, and a background process ensures that each target node learns it. The /Set command can be used through the network to modify the active/inactive status of the funclets, to change the lifespan of the funclets, or to delete funclets.

IV. IMPLEMENTATION FOR SENSOR NETWORKS

A. Software architecture

The CSP Virtual Machine (CSP-VM) was implemented in TinyOS [1], for MicaZ motes. The architecture of the system is shown in Fig. 2. The Funclet Manager handles user commands and automatic funclet lifecycle management. The Funclet Manager also adds new funclets to the Funclet Table and removes deleted funclets from the Funclet Table. Additionally, the Funclet Manager handles the active status of the funclets. The Event Manager handles received events (radio messages, sensor measurements, and timer events) and stores them in the Event Queue. The Execution Manager evaluates events and sets the triggered status for the corresponding funclets. This block contains the scheduler as well. The Interpreter executes the funclet, which is in Running state.

Note that there are two blocks which can receive events from the environment. Event Manager handles system events while Funclet Manager handles user commands. These blocks run virtually in parallel, and they can interrupt the Execution Manager and the Interpreter, but apart from the delay caused, they have no effect on the interrupted operation.

The Event Manager inserts any received event into the Event Queue. When no funclet is actually triggered, the Execution Manager checks whether the latest event in the Event Queue matches the external event part of any of the active funclets. Matching funclets are set to Triggered state; if no funclet was triggered, the event is removed from the Event Queue. If there is a funclet in Triggered state, the Execution Manager sets one of them to Running state and calls the Interpreter. The Interpreter reads the triggering event’s data and then executes the internal events of the funclet.

The automatic activation and deactivation of a funclet is controlled by the lifespan of the funclet, which is defined when the funclet is learned. The lifespan data contains the activation and deactivation time of the funclet. Also, direct management commands can be used to immediately activate or deactivate funclets.
When the Interpreter finishes the execution of a funclet, the Execution Manager looks for further triggered funclets and calls the Interpreter to execute them. When no more funclets are triggered, the Execution Manager removes the triggering event from the Event Queue and processes the next event (if any).

B. Evaluation

Table I compares the memory footprint and the transmitted code size for three different implementations of an example published in [4] – a native TinyOS implementation, an ASVM implementation [4], and a CSP-VM implementation. (For ASVM syntax, refer to [4]. For CSP-VM syntax, see the Appendix.) In the example, a light sensor is read, and the measured value is broadcasted through the radio channel. Note that the code sizes of the virtual machines are not much larger than the native code size of this simple problem. When the motes running native code are reprogrammed, the full program and operating system must be transmitted. Using virtual machines, the transmitted code size is three orders of magnitude smaller.

The memory footprints for the CSP-VM and the ASVM are comparable. The CSP-VM has somewhat larger program size, due to the higher level of abstraction in the programming language. Fig. 3 shows the memory footprint of the CSP-VM on the MicaZ platform. Note that almost 3 kB are available for storing funclet code.

<table>
<thead>
<tr>
<th>Table I. MEMORY FOOTPRINT AND TRANSMITTED CODE SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code (Flash)</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>22 kB</td>
</tr>
<tr>
<td>Data (RAM)</td>
</tr>
<tr>
<td>Transmitted program</td>
</tr>
<tr>
<td>Program code:</td>
</tr>
</tbody>
</table>

To test the overhead of the CSP-VM, three benchmark tests were run with both the CSP-VM and the native TinyOS code.

Benchmark 1

A transmitter node periodically broadcasts packets containing 10 bytes of data. The benchmark node receives the packet and sums its content. The benchmark node contains the following funclet:

```
Funclet1: c?m1,m1>K[],Y=0,f(i:10){Y=Y+K[i]}
```

Benchmark 2

A transmitter node periodically broadcasts packets containing an integer number from a set. The benchmark node receives the packets and calculates the frequency of the received numbers. The benchmark funclet is the following:

```
Funclet2: c?m1,m1>a,A[a]++
```

Figure 3. Memory footprint of the CSP-VM

BENCHMARK 3

A transmitter node periodically broadcasts packets containing an integer, and a benchmark node receives these packets. If the number is odd, the benchmark node increments the number and rebroadcasts it; otherwise the benchmark node drops the packet, using the following funclet:

```
Funclet3: c?m1,m1>a,i(a%2)<(a++,m1<,a,c1m1)
```

The execution times were measured using a digital oscilloscope, measuring signals on the motes’ LEDs. The benchmark CSP nodes turned on the red LED when the packet was received at T1, turned on the green LED when the processing of the corresponding funclet begun at T2, and turned off both LEDs when the processing was finished at T3. The native benchmark node similarly turned on the red LED when the packet was received, but here the processing started without delay. The LED was turned off when the processing was completed. T_{csp} = T_{2} - T_{1} is the execution time for the benchmark, while T_{native} = T_{2} - T_{1} is the scheduling/preparation time (zero for the native code). The execution time for the three benchmarks can be seen in Table II. The scheduling times were in the range of 2-2.1 ms, independent of the actual funclet under execution. As expected, the CSP-VM has a large overhead, resulting in an execution time of 44-270 times larger than that of the native code. Note that the measured CSP interpreter was not optimized for speed. Thus, a significant improvement would be expected in subsequent versions of the CSP-VM.

<table>
<thead>
<tr>
<th>Table II. EXECUTION TIMES FOR THE BENCHMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>Benchmark 1</td>
</tr>
<tr>
<td>Benchmark 2</td>
</tr>
<tr>
<td>Benchmark 3</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table III gives a comparison of the reprogramming overhead. For the native code, the full program and operating system must be transmitted when reprogramming the nodes,
The CSP-VM clearly has a much smaller reprogramming overhead, while its execution is two orders of magnitude slower than that of its native counterpart. However the CSP-VM could cause increase in power consumption, depending on the nature of the application.

V. CONCLUSION

A sensor networking architecture was proposed for reconfigurable distributed measurement systems. The system services are composed of lightweight functional elements, called funlets. The measurement system can be adapted to changing requirements by adding/replacing funlets. Due to the small size of the funlets, the reconfiguration is very efficient.

The proposed architecture was implemented for the MicaZ platform under TinyOS. The CSP-VM was compared to the native implementation and to the virtual machine of Active Sensor Network platform (ASVM). The CSP-VM architecture provided comparable results with the ASVM implementation, and was found to be much easier to reconfigure than the native code implementation. According to measurements, the execution of programs constructed from funlets on the current non-optimized implementation of the CSP-VM is two orders of magnitude slower than that of its native counterpart.

The proposed architecture is based on Communicating Sequential Processes and thus provides the straightforward possibility of verification and checking of system correctness with the available verification tools.

ACKNOWLEDGMENT

This research was partially supported by the Hungarian Government under contract NKFP2-00018/2005 and the European Defence Agency under contract A-0829-RT-GC.

REFERENCES


APPENDIX - FUNLET SYNTAX

The funlet syntax was designed to provide an expressive set of commands to represent sensor networking problems, while at the same time keeping the complexity (and thus the memory footprint) of the virtual machine low.

funlet ::= external_event, internal_events
external_event ::= channel_read | manage_event
internal_events ::= internal_event [], internal_events
internal_event ::= channel_write | message_operations | assignment | flow_control

The general syntax of channel read is the following:

channel_read::= com_channel?message_variable | meas_channel?scalar_variable | timer_channel?scalar_variable

where the channel is the event source and the variable is the name of variable where the received message is stored. Specifically, handled channel types are the following: com_channel denotes the incoming communication channel, meas_channel is a measurement channel (e.g. light, temperature), and timer_channel is a timer. The semantics of channel read events are the following:
com_channel?message_variable receives a message of type identical to the type of message_variable and stores the content of the message in message_variable. (See below the handling of message variables.)

Notes:
1. Only the reception of the correct type of message triggers the event.
2. The communication channel has a high-level abstraction. Thus it can denote various physical or logical channels. For example, different radios can be used as different channels, or various routing algorithms can be preprogrammed and used as different channels.

meas_channel?scalar_variable stores the result received from the given measurement channel (e.g. light, temperature) in the specified variable. A measurement always has to be initiated by writing to the measurement channel (see channel write below), and the measured results can be read when the measurement event is activated (possibly after a delay required to perform the measurement). The virtual machine handles the low-level management of the sensors.

timer_channel?scalar_variable stores the local time of triggering in the optional variable.

manage_event ::= manage_type[?]

where manage_type is an event associated with funclet management activities (teach, activate, deactivate, delete). Note that this event is not followed by a variable. Funclets with a manage event can be used e.g. for initialization purposes.

Internal events are handled as follows:

channel_write ::= com_channel!message_variable
| meas_channel!scalar_variable
| timer_channel!scalar_variable

The write operation to a channel initiates message sending on the particular channel. The semantics of channel write events are the following:

com_channel!operand initiates the sending of message with content operand on com_channel. The VM automatically handles all low level communication details. Note that not only the content of the message is forwarded but its type, too.

meas_channel!scalar_variable starts the specified measurement, which will cause a corresponding measurement event upon completion. Note that the variable may be omitted.

timer_channel!scalar_variable sets the timer associated with timer_channel to periodic operation with time period scalar_variable, starting from the time instant of the write operation. The time is measured in millisecond. The write operation timer_channel!0 stops the timer at timer_channel.

message_operations ::= message-compose | message-decompose

message-compose ::= message_variable<
scalar_variable [ scalar_variable]*
message-decompose ::= message_variable>
scalar_variable [ scalar_variable]*

Message composition stores the listed variables in the given message variable in the specified order. The decompose operation decomposes a message into variables, based on the type (size) of the variables. The message variables may have an arbitrary length up to a maximal size. The message may be transmitted in multiple packets; the virtual machine handles low level details of slicing and reconstruction of the messages, if necessary. The arithmetic and logical operations are the following:

expression ::= unary_l_operator operand | auto_assign_operation | operand binary_operator operand
auto_assign_operation ::= operand
unary_l_operator ::= ! | - |
unary_r_operator ::= ++ | --
binary_operator ::= + | - | * | / | %
& | | | < | > | <= | >= | == | !=
operand ::= scalar_variable | constant
assignment ::= scalar_variable = expression | auto_assign_operation

The program flow statements are the following:

flow_control ::= choice | loop
choice ::= if(expression)(internal_events) ((internal_events))
loop ::= loop_long | loop_short
loop_long ::= for(assignment, expression, assignment){internal_events}
loop_short ::= for(variable: constant){internal_events}

The loop is very similar to the for statement in the C programming language, with an optional short form, which is equivalent to the long form of for(variable=0, variable<constant, variable++) {internal_events}.

variable ::= elementary_scalar | vector | message_variable
vector ::= array | associative_array
vector_element ::= array [ index ] | associative_array [ index ]
| associative_array [$ key ] |
| associative_array [# index ] |
| associative_array ![$ key ]
scalar_variable ::= elementary_scalar | vector_element

Elements of associative arrays may be referred to with an index (e.g. A[12]) or with a key (e.g. A[$446]). The key associated with an index may be returned (e.g. A[$12]), and also the time of the last assignment to an element may be returned (A![$12] or A![$446]). Index and key are scalar variables.