Rule Execution and Event Distribution Middleware for PROSEN-WSN

Xiang Fei Evan Magill
Department of Computing Science and Mathematics,
University of Stirling, UK.
xf@cs.stir.ac.uk ehm@cs.stir.ac.uk

Abstract

This paper presents a prototype wireless sensor network middleware REED (Rule Execution and Event Distribution). This middleware supports both the distribution of rules and the events that trigger them. REED employs a rule-based paradigm to allow sensor networks to be programmed at run time. This provides a flexible environment where applications and users can programme the sensor nodes to allow their behaviour to be changed at run time. Such a rule-based approach allows a number of services such as Subscribe-Notify to be constructed. A prototype system for PROSEN-WSN has been implemented which demonstrates the REED middleware. The main contribution of this paper is the ability to programme a rule-based WSN at run time. However we also illustrate the power of such rule-based programming on a working prototype. Our focus is on supporting the processing, filtering, and collating of data collected by a WSN.

1 Introduction

This paper describes a rule-based middleware for Wireless Sensor Networks (WSN) where the behaviour can be programmed at runtime. The work has been carried out in the setting of a wind farm. PROSEN[1.] (PROactive SENsors) is a multi-university project building the basis of a proactive wind farm condition monitoring system. It is well known that a WSN features a (large) number of (heterogeneous) embedded sensor devices, each of which has constrained processing power, memory and energy, and error-prone wireless links over which the devices communicate. This is a challenging environment for software development. In PROSEN, emphasis is given to a proactive approach to managing the collected data. In particular the network acts to provide pertinent filtered data. It also employs AI-based data analysis and a proactive goal-driven policy server. However this paper will focus on the programmability of WSNs. The middleware provides application developers with a suitable abstraction by employing a rule-based paradigm. The introduction of such middleware can provide a uniform programming environment to the application developer yet shield them from the complexities arising from a WSN.

The challenges from the WSN middleware are mainly the high integration with the physical world, a high degree of dynamics due to the environment and the system itself, and the limited available resources[2.]. As a result, the PROSEN WSN middleware is:

1. event triggered and light-weight, so that it can be running on a node with limited CPU power, memory and bandwidth;
2. programmable at run time, i.e. the system behaviour can be programmed by applications at run time so as to be adaptive to the applications' goals, and the changing environment;
3. capable of event subscription and notification, and hence only those requiring data will receive it. This significantly saves bandwidth and energy;
4. supports energy-conservation; e.g.. operates with different activity rates, i.e. sleep and work modes.

In this paper, we proposed a Rule Execution and Event Distribution (REED) middleware for PROSEN-WSN. This middleware supports both the distribution of rules and the events that trigger them. REED employs a rule-based paradigm to allow sensor networks to be programmed at run time. This provides a flexible environment where applications and users can programme the sensor nodes to allow their behaviour to be changed at run time. Such a rule-based approach allows a number of services such as Subscribe-Notify to be constructed. REED middleware is also lightweight and energy-conservative. In Section 2, the REED middleware architecture is described, followed by the definition of the formal language for REED. The REED middleware is evaluated in section 3. The rule management is also discussed in this section. A prototype implementation is described in section 4 to demonstrate the REED middleware.

Related work is discussed in Section 5, followed by the conclusion in section 6.

2 REED middleware architecture

2.1 General architecture

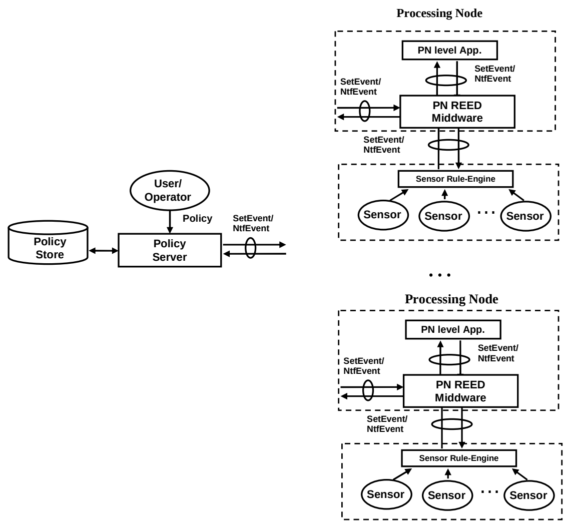

Figure 1. PROSEN system architecture

Figure 1 shows the system architecture for PROSEN, which consists of a Policy Server (PS), a PN (Processing Node) for each wind-turbine, and sensors to measure parameters such as temperature, wind-speed, wind-direction, battery-level, and gearbox temperature. The PS interacts with users and operators to obtain the goals for the system. Such goals might describe a desirable power output or response to poor weather conditions. The PS converts the goals to a set of policies. These policies in turn are converted to low-level rules. These rules describe the behaviour of individual PNs. Hence the WSN distributes and executes these low-level rules within each PN. It is also possible to transfer these rules between PNs.

In addition to transferring the rules, the REED middleware also transfers events between the system components. It is these events that trigger the individual rules.

Conceptually, a *rule* takes the form of <*event*, *condition*, *action*> where:

- an *event* is received from any other component in the system. This is often an event carrying data values, but other events such as a timeout event, a sleep or wake-up event can also occur.
- a *condition* is a Boolean expression that will be evaluated when the *event* occurs.
- an *action* is executed if the above *condition* is true when the *event* is received. The action may manipulate or store data. It may also generate another *event* to other components in the system, such as an *event* to trigger other *rules*.

To implement REED, a *rule-engine* has been designed and implemented. The functionality of the *rule-engine* includes:

- managing a *rule-base* that stores the *rules* for the middleware to allow the adding, removing, and overriding of rules
- verifying *rule* consistency, and
- executing the *rules* in response to received *events*.

Figure 2 shows the general architecture of the REED middleware. This echoes typical structures given in the literature. The middleware must record certain aspects of the state of the node and the events that have occurred. These are recorded in the *Fact-Base*. Here we borrow the terminology *Fact* from a separate rule-based WSN approach[11.]. The *Event-Manager* is responsible for receiving events, passing them to the *Rule-Engine*, where the engine executes any matching rules, and distributes any resulting events. The *Rule-Base* stores all the rules used by the engine.

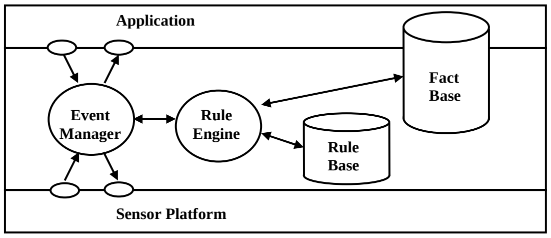

Figure 2. REED architecture

The REED middleware actually has two levels of rule-engine within a PN. Figure 1 illustrates the two-level REED architecture for PROSEN, where the *sensor rule-engine* is responsible for local sensor data collecting and processing, while the *PN REED middleware* is employed for wider event processing; such as data event correlation between processing nodes. In this paper we shall focus on the PN level REED middleware and this is discussed further in Section 3.

2.2 Language definition

In order to provide a clear description of the REED middleware, we will use a formal notation. The notation is explained in Table 1 and gives the core definitions. Indeed this notation is used within our implementation.

Table 1. Core language definition for REED

Property = <PropertyName "=" PropertyValue>
State = <StateID ";" Property | State ";" Property>
Event = < EventID ";" Property | Event ";" Property >

```
FactID = < StateID | EventID >
Fact = <State | Event>
ComparisonOperatior = < ">>" | "<<"| ">="| "<="|
"= ="| "!=">
Connector = < "&&">
ExistOperator = < "∃" >
Condition = <EsistOperatpor "(" FactID ")"| FactID
"." PropertyName ComparasionOperator Threshold |
FactID "." PropertyName ComparasionOperator
FactID "." PropertyName >
ConditionSet = <Condition | Condition Connector
Condition>
Action = < Store "("Fact")" | Send "("Destination ","
Event")" | FunctionCall "("Event ")" | …>
ActionetSet = < Action | Action "," Action>
EventHandler = < "(" ConditionSet ";" ActionSet ";"
Priority ")" | EventHandler ; "(" ConditionSet ";"
ActionSet ";" Priority ")">
Rule = <Event_ID ";""["EventHandler"]" >
```

3 Evaluation of REED

3.1 Programming at run-time

Rule-based middlewares, such as to FACTS[10.], enable individual WSN nodes to be programmed. The stored rules capture the expected behaviour resulting from certain events, conditions or states. However the rules do not change once they have been deployed and stored. In contrast, for PROSEN the PS alters the low-level rules after deployment so it is important to allow the rules stored within the PN to be changed at any point in time. In other words, it is required that the system can be programmed at run time.

Another advantage of providing a dynamic rule-set is the ability to easily apply REED to other applications without the need to re-flash the static rule-set within a PN. However to support dynamic updates of the *rule-base*, rule management is required and Section 3.3 will discuss this in more detail.

3.2 Support for subscription-notification

Programming a PN with REED low-level rules allows a broad and flexible approach. For example an application may want to provide an event subscribe-notify service as part of its solution. Here we show that this can be constructed using our low level rules. In effect the sending of a rule acts as a subscription, and the triggering of the rule acts as a publication. Of course the rule action must send the event back to the source of the rule; i.e. the subscriber. (To do so is the choice of the source of the rule; our approach does not require that any generated event returns to the rule source.) Consider an example; a PS subscribes to a *PN REED middleware* by sending a rule to a particular PN as follows:

Rule = <*max_wind_speed*; [(*max_wind_speed*.*value* >> 60*; send* (*PS*, *maxWindSpeed*))]>

(For simplicity, the *Priority* field is not included in the *EventHandler* thereafter.)

Later on, the REED receives an event from its *sensor rule-engine* as follows:

maxWindSpeed = < *max_wind_speed*; *Value* = 67; *Time* = 23:14:12; *Date* = 01-02-08>

This event will trigger the execution of the rule above, and as a consequence, this event will be notified to the PS. There can of course be more than one subscriber; i.e. an event results in a number of notifications.

3.3 Rule-base management

In REED middleware, the *rule-base* is used to store the rules for the *rule-engine*, and as such, the *rule-base* management is a major task of the rule-engine. In addition to the adding, removing, or overriding rule functions, the rule-engine should be able to maintain the consistency of the *rule-base*, and allow rules to be merged and filtered. This is because the rule-engine may receive the rules set from various sources. In PROSEN, the PN level REED middleware may receive the rule set from the authorised policy server, its own application entities, or from its authorised peers.

3.3.1 Rule-base consistency. To maintain the consistency of the rule-base, the rule-engine should detect and resolve any conflicting rules. These conflicts arise as an event may trigger more than two rules and generate conflicting actions, e.g. one rule setting a sensor on and another rule setting the same sensor off. Normally, the way to resolve this is to set different priorities so only the rule with the highest priority will be triggered. In PROSEN, the control related rules from the PS are given priority over event rules from other sources. As the PS uses meta-policies to maintain consistency, we avoid any inconsistency within a PN. We will develop a stronger mechanism for the PN in the future.

3.3.2 Rule merge and filtering. In PROSEN, the PN-level REED middleware accepts rules, and forwards any sensor-level rules to the *sensor rule-engine*. As the *sensor rule-engine* runs on a more resource-limited processor, the forwarded rules should be filtered and merged to remove any redundancy. For example, the REED on a PN receives a rule from the PS indicating:

< *max_wind_speed*; [(>> 70; *send*(*PS*, *maxWindSpeed*))] >,

and also receives a rule from another PN (denoted as PNx) saying:

< *max_wind_speed*; [(>> 50; *send*(*PNx*, *maxWindSpeed*))]>.

Instead of sending two corresponding rules to the *sensor rule-engine*, the REED sends only one rule

<*max_wind_speed*; [(>> 50; *send*(*REED*, *maxWindSpeed*))]>

to the *sensor rule-engine*. When a *maxWindSpeed* event is sent from the *sensor rule-engine* to the REED, the *REED rule-engine* will, based on the real wind speed reading, first check whether the wind speed is greater than 50 mph, and then check whether the wind speed is greater than 70 mph, to determine where to send the notification; to the PN only if the reading is between 50 and 70, or to both the PS and the PN if the reading is over 70.

For space considerations we simply give a brief description of the rule merge and filtering algorithm for PROSEN in Table 2, and suppose the *ConditionSet* contains one *Condition*.

Table 2 Algorithm for rule merge and filtering

Definition 1: Given a *Condition1* and a *Condition2*, $\forall$ *event*, if in meeting *Condition1* means it also meets *Condition2*, then we say *Condition1* is covered by *Condition2*, denoted as *Condition1* $\subseteq$ *Condition2*

Definition 2: Given

rule1 = <*eventID*, [(*Condition1*, *ActionSet1*)]> and

rule2 = <*eventID*, [(*Condition2*, *ActionSet2*)]>,

if *Condition1* $\subseteq$ *Conditiont2*, then we say *rule1* is covered by *rule2*, denoted by *rule1* $\subseteq$ *rule2*, which means that if *rule1* is triggered by *event*, the *rule2* must be triggered too.

Suppose the *PN REED rule-engine* receives a rule <*event_ID*, [(*Condition1*, *ActionSet1*)]>, denoted by *R1*, where *event*, with its identifier being *event_ID*, can be generated from the *sensor rule-engine* and there is no other rule in the current *rule-base* that has coverage relationship with *R1*, the REED will save this rule to the *rule-base*, construct a rule < *event_ID*, [(*Condition1*, *send(REED*, *event*))]> and forward this rule to the *sensor rule-engine*. Later on, the *rule-engine* receives another rule <*event_ID*, [(*Conditions2*, *ActionSet2*)]>, denoted by *R2*,

- If *R2* is covered by *R1*, the *rule-engine* is not going to forward this rule to the *sensor rule-engine*, instead, it changes the *R1* originally saved in the *rule-base* to <*event_ID*, (*Condition1*, *ActionSet1*) $\rightarrow$ (*Condition2*, *ActionSet2*) >, where the symbol "$\rightarrow$" means a coverage link with (*Condition1*, *ActionSet1*) being the head and (*Condition2*, *ActionSet2*) being the tail of the link. In addition, each node in this coverage link will be accompanied by a counter with the initial value being 1. So if the same rule is received, the *rule-engine* simply increases the counter for that node by 1.
- If *R2* covers *R1*, the *rule-engine* will change the *R1* originally saved in the *rule-base* to <*event_ID*, (*Condition2*, *ActionSet2*) $\rightarrow$ (*Condition1*, *ActionSet1*) >, construct a new rule < *event_ID*, [(*Condition2*, *send*(*REED*, *event*))]>, and forward this rule to the *sensor rule-engine* to replace the original one.

When the *rule-engine* later on receives a rule <*event_ID*, [(*Condition3*, *ActionSet3*)]>, denoted by R3, and the current coverage link for *event* is (*Condition1*, *ActionSet1*) $\rightarrow$ (*Condition2*, *ActionSet2*), then the (*Condition3*, *ActionSet3*) will be inserted into this coverage link, and then updates the rule to the *sensor rule-engine* if it becomes the new head of this coverage link.

When a *Remove*(*R1*) is received, the *rule-engine* will check whether (*Condition1*, *ActionSet1*), is at the head of the covering link.

- If it is not at the head, the *rule-engine* firstly decrements the counter for (*Condition1*, *ActionSet1*) by 1, and if the result reaches 0, this node will be deleted from the link.
- If it is at the head, and its counter had the value 1, the covering link will be updated by deleting (*Condition1*, *ActionSet1*) and check whether it has a child node.
    - If it has no child node, the rule-engine will send a command to the *sensor rule-engine* to delete the rule <*event_ID*, [(*Condition1*, *send*(*REED*, *event*))]>.
    - If, say (*ConditionSet2*, *ActionSet2*), is the child node, this node will become the head of the coverage link. The *rule-engine* then construct a new rule < *event_ID*, [(*Condition2*, *send*(*REED*, *event*))]>, and forward this rule to the *sensor rule-engine* to replace the original one.

3.4 A light-weight middleware

REED is lightweight in terms of the energy and memory consumption. This is because first of all, it is event triggered instead of continuously polling and this saves wireless bandwidth and energy. Secondly, unlike JESS[14.] where all the facts are stored in its working memory before the execution of their rules, REED filterers the received data events using its rules and only those needing further processing will be saved to the *fact-base*. This makes the overhead for memory consumption much lower. Thirdly, the subscribe-notify service (or indeed any rules only generating a data anomaly) ensure that data events only go to those components that require them. This is in contrast to LIME[5.] and TinyLIME[6.] middleware which are Tuple Space-based where the data sharing and synchronization across the network is both bandwidth and CPU consuming.

For real applications, some rules can be set as default rules and are put to the REED rule-base locally during the initiation. The rules are updated at run time only when necessary. This will further save the power for rule distribution and rule management.

As REED is event based, it can go into a *sleep* state in order to save the battery energy when there is no event for processing. It returns to the *work* state either by a scheduled timeout or a triggered event from a lower level source. In PROSEN, the signals for *sleep* and *wake-up* are triggered by the *sensor rule-engine* which is always in a *working* state. When the *sleep* event is received, the REED writes the unsaved rules and necessary facts to the flash memory before it exits. When the REED is initiated as the result of the *wake-up* event, it will, before processing any event, restore those rules and facts back from the flash memory.

4 Prototype implementation

4.1 Prototype implementation architecture

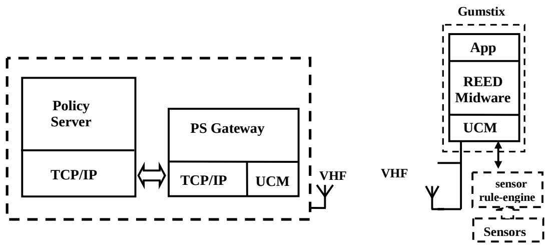

Figure 3: Prototype implementation architecture

Figure 3 shows the prototype implementation architecture. The software structure for REED middleware is illustrated in Figure 4. REED sends and receives external messages via the interfaces provided by the UCM. The UCM (Unified Communications Manager), developed by other partners in the PROSEN project, provides a platform to communicate via wireless links such as VHF channels. The *Event Constructor* constructs events with the received messages. It classifies them either as *SetEvents*, (rules), or as *NtfEvents* (e.g. data events), and then puts them onto their corresponding queues. These two queues may have different priorities. When any event is to be distributed, the *Msg Constructor* will transform it to the corresponding message format before delivering it to the UCM.

REED is running on a Gumstix™[12.] GS400K-XM, which is a tiny full function Linux motherboard based on low power Intel XScale® technology. (Later we plan to port this to a specialised processor supporting a JVM.) GS400K-XM has 16MB flash memory which can accommodate JamVM[13.], which is a compact JVM (Java Virtual Machine), and so REED is developed using Java.

At the time of writing, the core functionality of REED has been implemented; that is, functions for adding and updating rules, executing rules triggered by events, merging and filtering rules based on one condition, and *rule-base* and *fact-base* store/recovery for *sleep* and *wake-up* events. Although a sensor rule-engine has been built, here we employed a simple event generator to emulate the *sensor rule-engine*. Rules and events such as those shown in section 3 have been used to test the rule management, and test the rule execution. A distributed sensor data aggregation algorithm has also been implemented using REED. This work will be described in another paper.

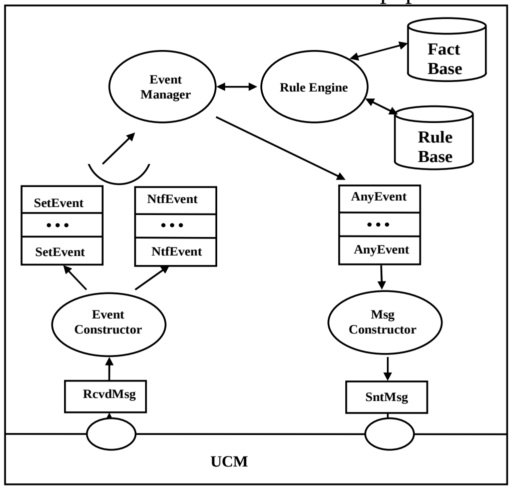

Figure 4 REED Software Structure

5 Related work

[3.] and [4.] provided surveys across a broad array of WSNs and middleware. Among those available mechanisms, we found the LIME [5.] (Linda in a Mobile Environment) and TinyLIME [6.] based solutions attractive. LIME and TinyLIME provide a Tuple Space based middleware. However, LIME is heavy-weight in that mobility management and data synchronisation are bandwidth and CPU consuming. TinyLIME is the extension of LIME, but it cannot be employed directly on currently available sensor processing nodes such as Tmote. A special interface has to be provided to bridge TinyLIME running on the base station and applications running on sensor nodes.

[8.] proposed an event-based distributed middleware architecture, Hermes, that follows a type- and attribute-based publish-subscribe model. In [7.], SIENA, an event notification service consisting of notification selection service and notification delivery service has been presented. SIENA exhibit both expressiveness and scalability. However, both Hermes and SIENA are for IP based Internet.

[9.] proposes an ECA (Event, Condition and Action) rules based middleware model for WSN. However, no evaluation or prototype implementation is described. In [10.], a rule-based middleware architecture for WSN, called FACTS, was proposed, and [11.] described its programming primitives and implementation using the Haskell programming language. Compared to FACTS, our proposal is distinctive in the following ways: first, the rule set in FACTS is static while the *rule-base* in REED is dynamic as the rules for REED middleware can be updated at run time. Furthermore, the REED prototype has been implemented, which demonstrates not only the functionality but also the usability of REED.

JESS is a rule-engine written entirely in Sun's Java language[14.]. It is for general purpose and not dedicated for WSN environment. As a consequence, the memory usage is not optimized[10.] for running on sensor nodes. In addition, in JESS, all the facts are stored in its working memory before executing the rules while in REED, any received data event will be filtered by rules first and only those needing further processing will be saved to the *fact-base*. As a result, the overhead for memory consumption is much lower than using JESS.

6 Conclusion

In this paper, the REED middleware is described. It supports both the distribution of rules and the events that trigger them. REED employs a rule-based paradigm to allow sensor networks to be programmed at run time, so that applications and users can programme the sensor nodes to allow their behaviour to be changed at run time. Such a rule-based approach allows Subscribe-Notify service to be constructed. To support this programmability, the rule management is also discussed. The prototype implementation demonstrates the REED middleware. So far, the integration of REED middleware with the PS and the UCM has been performed. Currently the complete PROSEN system integration and deployment is being carried out. In the future, more domain knowledge will be collected and expressed via rules for REED to support processing, filtering and collating data within the wind farm setting.

7 Acknowledgements

The authors would like to thank EPSRC (Engineering and Physical Sciences Research Council) for the funding the PROSEN project. We are also indebted to our PROSEN research colleagues for their support.

8 Reference

[1.] PROSEN: http://www.prosen.org.uk/

[2.] K. Römer, O. Kasten, and F. Mattern, “Middleware Challenges for Wireless Sensor Networks, Mobile Computing and Communications Review”, Vol. 6, No. 2, 2002

[3.] M. Kuorilehto, M. Hannikainen, and T. D. Hamalainen, “A Survey of Application Distribution in Wireless Sensor Networks”, Journal on Wireless Communications and Networking 2005:5, 774–788

[4.] E. Yoneki, and J. Bacon, “A Survey of Wireless Sensor Network Technologies: research trends and middleware’s role”, Technical Report www.cl.cam.ac.uk/techreports/ UCAM-CL-TR-646.html , 2005

[5.] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman, LIME: A coordination model and middleware supporting mobility of hosts and agents: Vol 15 , Issue 3, pp: 279 – 328, July 2006

[6.] A. L. Murphy and G. P. Picco. “TinyLIME: Bridging Mobile and Sensor Networks through Middleware”, PERCOM (Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications), pp: 61 – 72, 2005

[7.] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, “Design and Evaluation of a Wide-Area Event Notication Service”, ACM Trans. on Computer Systems, 19(3):332-383, Aug. 2001.

[8.] P. R. Pietzuch and J. M. Bacon, “Hermes: A Distributed Event-Based Middleware Architecture”, In *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611-618, Vienna, Austria, July 2002.

[9.] C. Zhang, M. Li and Q. Pan, “An ECA Rules Based Middleware Architecture for Wireless Sensor Networks”, Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT), Pages: 586 – 588, 2005

[10.] K. Terfloth, G. Wittenburg; and J.Schiller, “FACTS - A Rule-Based Middleware Architecture for Wireless Sensor Networks”, First IEEE International Conference on Communication System Software and Middleware (COMSWARE 2006), New Delhi, India, January 2006

[11.] K. Terfloth, G. Wittenburg; and J.Schiller, “Rule-oriented Programming for Wireless Sensor Networks”, International Conference on Distributed Computing in Sensor Networks (DCOSS) / EAWMS Workshop, San Francisco, USA, June 2006

[12.] Gumstix: http://gumstix.com/

[13.] JamVM: http://jamvm.sourceforge.net/

[14.] JESS, the Rule-Engine for the Java platform, http://www.jessrules.com/jess/index.shtml