On Coordination Tools in the PicOS Tuples System

Benny Shimony
Computing Science Dept.
University of Alberta
Edmonton, Alberta
Canada T6G 2E8
shimony@ualberta.ca

Ioanis Nikolaidis
Computing Science Dept.
University of Alberta
Edmonton, Alberta
Canada T6G 2E8
nikolaidis@ualberta.ca

Pawel Gburzynski
Olsonet Comm. Corp.
51 Wycliffe Street
Ottawa, Ontario
Canada K2G 5L9
pawel@olsonet.com

Eleni Stroulia
Computing Science Dept.
University of Alberta
Edmonton, Alberta
Canada T6G 2E8
stroulia@ualberta.ca

ABSTRACT

In this paper, we discuss the most recent *coordination* extension to the PicOS-tuples environment, inspired, to a degree, by B-Threads and FACTS. We illustrate the extensions with two design patterns, highly useful in WSN computations, known as *regulative superimposition* and *distributed detection*. Those patterns are employed in a debugging protocol that retrieves snapshots of node states. We demonstrate how our new idioms can be propitious for separating concerns in WSN programming using tuples.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures, languages, patterns*

General Terms

Design

Keywords

Superimposition, coordination interface, tuple space

1. INTRODUCTION

The task of programming sensor-based systems can be quite complex and intricate, calling for an experienced programmer with an integrated view of all levels of the target system [1]. On the one hand, sensor nodes are required to respond in a timely fashion to complicated configurations of events, with multiple events potentially occurring at (almost) the same time. On the other, the same nodes are forced to operate within a resource-constrained framework (memory, CPU power, and energy) which is additionally plagued by inherent reliability problems (whimsical connectivity, interference, mobility).

In this setting, the event-based programming paradigm has been the prevailing leitmotif of the popular programming platforms for WSN nodes [2]. This paradigm has been demonstrated to stimulate efficient solutions in terms of energy usage, memory footprint, and concurrent task management. It is also congenial for coping with the reliability problems, as it facilitates various persistent and idempotent algorithms which can be naturally harnessed to reaching complex goals through congregations of repetitive smaller steps. Unfortunately, in many practical systems, the above benefits come at the detriment of qualities, such as understandability, maintainability, and extensibility of code [5, 16].

To mitigate these issues, software-engineering techniques are needed to impose a more modular structure to the programs in a way that does not compromise the primary advantages of the original paradigm, especially the memory footprint, the computational complexity, and the energy requirements.

Our group has been working with PicOS, a low-overhead operating system for wireless sensor nodes that merges the traditional event-based sensor-programming paradigm with a finite state machine (FSM) abstraction. PicOS-tuples is a recent extension of PicOS, inspired by Linda [8], whose conceptual simplicity and ability to address systems with unreliable communication channels and processing units has stimulated several solutions for WSNs. In this paper, we propose a set of new coordination constructs, designed to further improve the high-level characteristics of PicOS WSN programs without any additional constrains on their resource budget.

The rest of this paper is organized as follows. Section 2 presents two common WSN programming examples and Section 3 reviews the concept of *superimposition* and its variants. Section 4 presents the PicOS operating system for WSNs. Section 5 presents the new coordination idiom and compares it to the B-Threads platform. Section 6 reviews the application of this coordination idiom in the context of detailed programming examples. Section 7 includes a brief discussion and relates our work to related work. Finally, Section 9 concludes by summarizing the state of our work

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SESENA '11, May 22, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0583-9/11/05 ...$10.00.

and outlining future directions.

2. WSN PROGRAMMING EXAMPLES

In this section, we discuss two interesting problems that WSNs have to address, potentially with superimposition coordination approaches. Let us consider first the "broadcast storm" problem in WSN [21], surfacing when the network must convey a broadcast message to all nodes. The standard first step in containing the flood of retransmissions is to cache received messages (or their fingerprints) and filter out duplicates, such that they are neither received nor subsequently retransmitted. With specific instances of the problem, the filtering criteria can be conceivably more subtle (e.g., to not reject all duplicates as a mechanism to counteract flaky wireless reception). The fact that these criteria are orthogonal to the node's application code implies that they should be maintained separately from the proper application code, and should be accessible by a separate code layer *inspecting* and *regulating* the base code.

Another challenge in WSNs comes with the system deployment phase where the ultimate reality check often mercilessly exposes undesirable behaviors, not observed during the more sterile, artificial testing environments. While powerful and authoritative virtual testing environments [4] can go a long way towards spotting and eliminating complex (distributed) bugs in pre-deployment tests, they cannot possibly locate all bugs, most notably those resulting from hardware errors, misspecification, or subtle real-time race conditions. Consequently, a robust WSN system should come with a built-in debugging capability, available post-deployment. This requirement is known as the *protocol visibility* postulate [15] and boils down to adding to the original code of the application assertions and snapshot triggers that would enable the operator to collect the distributed state of the system. That extra code will be dormant during normal operation, but the debug utility of the operations support system (OSS) should be able to activate it upon demand. By definition, a debugging system built along this line consists of *regulation* code being *imposed* on the base code.

3. SUPERIMPOSITION

As a formal concept, *superimposition* [7] allows for a layered way of extending the functionality of an existing code base. Each additional layer defines the conditions under which new concerns become relevant. Note that this operation is non-commutative, i.e., the added layer has a controlling role over the preceding layers (which is clearly not reciprocal). For this reason the new layer is called the *regulator*: its regulative role applies to all the preceding layers. The concept was first demonstrated by Dijsktra et al. [6] and was further developed, e.g., in [7, 10, 12]. Based on the character of the control exercised by the regulator, the following classification has been put forward.

Spectative superimposition is the weakest (least invasive) superimposition form. In this variant, the regulator can only add computations that modify its own variables, based on (passive) reference to the underlying program. The regulator cannot modify or affect the behavior of the basic program in any way. This approach preserves the safety and liveness of the basic program. This kind of superimposition has been adopted in the UNITY language [9].

Regulative superimposition additionally allows the regulator to restrict the basic program, by either delaying or blocking certain operations [7]. Semantically, it can be viewed as a way of strengthening the guard conditions of the basic code. This approach only preserves the safety of the basic program, possibly compromising its liveness.

Invasive superimposition, the strongest form, allows fully intrusive regulation, whereby the superimposed code is permitted to modify the variables of the underlayer. Needless to say, this approach does not preserve the safety (not to mention the liveness) of the basic program.

In the examples presented in this paper the *regulative* form of superimposition was chosen as the coordination idiom, first because of its simple format and the preservation of safety properties. The *spectative superimposition* is too weak for blocking and halting the basic program operations, while *invasive superimposition* requires complicated constraints and proof assertions to preserve the basic properties of code. A more detailed and formal account of superimposition (also known as *superposition*) and its refinement rules in the context of action-based systems can be found in [14]. The context of action-based systems is of interest to us, since we claim that PicOS execution semantics fall into this category.

4. PICOS IN CONTEXT

The basic assumption of an event-based language is that program execution can be expressed as an interleaved series of *actions* carried out in response to *events*. It is often convenient to assume that the actions are atomic: once started, an action is not preempted by another action until completed. This assumption eases the problem of reasoning about concurrency and simplifies the implementation. Most notably, it allows the implementation to be extremely frugal with respect to memory, which brings about important advantages in the context of resource constrained WSNs. PicOS [3], B-Threads [11], and FACTS/Linda [19, 8] are three such languages.

4.1 PicOS praxes and states

PicOS is an operating system for small-footprint embedded applications designed to facilitate event-driven programming for memory-constrained devices. PicOS applications, dubbed *praxes*, consist of collections of threads, each one structured as a finite state machine (FSM). A single thread can describe a number of different event-response actions, associated with its *states*. The different states of a thread tag its *entry points* (e.g., see the statements in lines 4, 7, and 12 in Table 1) which are the only points where the thread's execution may commence.

Cooperative multi-threading is carried out in a coroutine-like fashion, with the preemptibility grain equal to the state-action of a thread. Once a thread begins execution in one of its states, the CPU remains assigned to the thread until it completes the current action (by hitting the end of the FSM function, or executing **release** or **proceed** – see lines 6, 11, and 14 in Table 1).

As threads can only be preempted by other threads at state boundaries and by hardware interrupts executing on the same stack, all activities in the system can share the same stack. Note that with **proceed** (line 14 of Table 1), the thread sets its new entry point to S_INIT. Formally, this happens instantaneously. However, the transition is viewed as an event response, which means that the thread *yields*

```
/*
_* Sensor Temperature .
_*/
   define tuple temperature [nid owner, int sessId, int value];
   define tuple temperature_req [nid source, int sessId];
```

```
1  fsm sensorListener 30 (byte *context) {
2  define context  temperature_req;
3  int temp_sample;
4  state S_INIT:
5     when_tuple (temperature_req, filter_func(), S_REPLY )
6     release;
7  state S_REPLY:
8     temp_sample = read_temp();
9     .....
10    delay (gen_rnd_timer(1000), S_TX);
11    release;
12 state S_TX:
13    send_tuple(temperature, "%w", temp_sample);
14    proceed S_INIT;
15 }
```

Table 1: PicOS FSM Thread Code

when it reaches **proceed**, and another thread *could* execute before the new state is entered (the respective action commences).

To remain alive, a thread must dynamically specify its transition function (the succession of states) in every action. In addition to the trivial and unconditional transition via **proceed**, a thread may issue one or more *wait requests*. Each wait request specifies (a) an event (for which the thread wants to wait) and (b) the state in which the thread will enter when the event occurs. For example, the action at S_INIT in Figure 1 includes one wait request, namely the operation **when_tuple** (in line 5). Here, the thread declares that it wants to be resumed in state S_REPLY upon the occurrence of an event represented by the tuple *temperature_req*. With this operation, following the completion of its present action, the thread will be awakened in state S_REPLY when a new tuple shows up in the node's view (i.e., arrives from the network or gets deposited in the node's local repository). Another example of a wait request is the **delay** operation at state S_REPLY (line 10). In this case, the event will be delivered by a timer going off after the specified number of milliseconds (the first argument of delay).

A single thread may await multiple events, i.e., it is allowed to execute multiple wait requests in any given action, which are interpreted as alternative waking conditions. In this case, the first of the awaited events to occur wakes the thread up, in the state associated with the event by the corresponding wait request. Notably, whenever a thread is awakened (its state action is scheduled for execution), all the pending wait requests are erased. Therefore, with each action the collection of waking conditions must be specified from scratch.

4.2 PicOS tuples

Tuple operations have been added in PicOS to mimic the corresponding Linda operations, but they only apply in the context of the local repository of a single node. Nevertheless, the local repository is also populated by tuples received as tuple transmissions of neighboring nodes. The full list of operations brought in by PicOS tuples is given in Table 2. The bottom part of the table details PicOS's FSM coordination constructs (not related to Linda), which will be discussed later.

```
<tuple_type> get <tuple_name> (field_x == value|'*', ... )
<tuple_type> remove <tuple_name> (field_x == value|'*', ... )
<tuple_type> set <tuple_name> [field_x = value| ⊥ , ...]
<tuple_type> tuple_send <tuple_name> [ field_x = value| ⊥ , ... ]

define context{ tuple_type, ... }
when_tuple (tuple_type, bool (*filter_func(..)), next_state )
watch_when (tuple_type, bool (*filter_func(..)), next_state )
block_tuple (tuple_type|tuple_instance, bool (*block_cond(..)) )
block_tuple (threadId, thread_state, bool (*block_cond(..)) )
```

Table 2: PicOS tuple operations

The format used by PicOS-tuples to reference tuples in the repository involves a similar kind of associativity as in the original Linda operations. The **get** and **remove** operations correspond to Linda's **rd** and **in**. They accept templates to be matched to tuples: each field can either provide a specific required value or contain the wildcard **'*'**.

Operation **set** corresponds to Linda's **out** and inserts a tuple into the repository. Each field of the new tuple can either provide a valid (assigned) value or be undefined '⊥' (which is the default). With **send_tuple**, which can be viewed as an extended variant of **set**, the program inserts the tuple into the local repository and, in addition, broadcasts it as a message over the RF channel. This is the way to share data with neighboring nodes.

This scheme of data exchange bears many similarities to the FACTS system [19]. FACTS, too, is a data-centric rule-based language, whereby data exchange involves named tuples (called *facts*), with each rule responding to some data-related events, and a (local) repository of facts. While PicOS tuples and FACTS have been similarly motivated, their language syntax, semantics, and implementations differ quite drastically. Rules in FACTS are high-level entities compiled to run over a middleware layer. The PicOS-tuples scheme, on the other hand, is strongly integrated with PicOS threads de facto becoming a component of the PicOS kernel. A brief comparison of syntax and language between the two methods is presented in the following sections.

5. PICOS COORDINATION

The PicOS execution model bears many similarities to the operation model of B-Threads [11]. A B-Threads program consists of *scenarios* operating independently. Each scenario resembles a state machine covering a sequence of events and responding to specific types of *event objects*. The B-Threads coordination abstraction utilizes a special *request-wait-block* interface synchronizing the event objects. Thread operations are interwoven by a central scheduler that produces an integrated behavior (selecting enabled operations in some prioritized fashion). PicOS FSM abstractions are similar to *scenarios*. Furthermore, the limited preemptibility mechanism of B-Threads (whereby a scheduled thread executes up to the next synchronization operation without preemption) resembles the scheduling of state actions in PicOS.

The coordination interface in B-Threads consists of three operations parameterized by event objects.

- **Request** a command execution, analogous to PicOS's wait request, but tentative: the specified event must be triggered for the execution to be scheduled, but it may be *blocked* (see below), in which case its occurrence

will be ineffective.

- **Block** events of a given type from triggering *Requested* actions. The triggering is deferred until the block is removed.

- **Wait/Watch** for an event which has been *Requested* by another thread. This is similar to *Request*, in that the issuing thread wants to carry out some action upon the occurrence of the specified event, but that action will only be triggered if the event occurs while being *Requested* by some other thread.

In order to implement such a coordination mechanism in PicOS, we added to it the analogues of **Block** and **Wait/Watch** operations. In the event based environment, the B-Threads Request resembles PicOS's waiting on an event trigger (through **when_tuple**), but works in a combination with an environment configuration which triggers external events (unlike the centralized scheduler of B-Threads). When the event is received (and is waited upon) the FSM proceeds to the requested state. Operation **watch_when** also amounts to a wait on the data event, but the action will only be triggered if there is another outstanding wait on the event.

The last operation, **block_tuple**, requires further consideration. In the Java-based B-Threads environment, blocking is parameterized over *event* objects and the special interfaces, which brings in the flexibility of blocking definitions over types, subtypes, etc. In the C-based environment of PicOS, we provide two possible interfaces. The first one blocks any other thread that waits on events of the same tuple type or tuple instance.[1] The second one blocks a specific thread and state described by the arguments. Both interfaces include a Boolean *block_cond* function parameter which can define arbitrary conditions qualifying the block operation.

6. EXAMPLES

In this section we revisit the two superimposition problems mentioned in Section 2 and we explain how the PicOS-tuples and its coordination extension address the corresponding challenges.

6.1 Duplicate Data Filtering

First, let us consider the problem of removing duplicate packets received during broadcast-based dissemination of data. In Table 3, we list a snippet of the FACTS program taken from [20] illustrating one implementation of that task. The rule *handleDataItems 30* (whose priority is set to 30) handles the events related to data reception. Whenever new data is received, the rule caches three attributes of the message: the source ID, the timestamp, and the type (those attributes are assumed to uniquely identify the data item).

In order to filter duplicate data, the rule *removeDuplicateDataItems 60* is added. Note that its priority is set to a priority higher than that of the previous FSM. An activation of the new rule is contingent upon two simultaneous conditions: data arrival (as in the case of the previous rule) as well as the presence in *dataCache* of a triplet matching the attributes of the received data item. Owing to the higher priority of the new rule, it will execute (removing the newly arrived data item) before the previous rule is given the opportunity to act on it.

Table 4 shows how a similar functionality is implemented within the PicOS Tuples system. In state DD_INIT, thread *handleData* waits on the event consisting in the arrival of a new data tuple (line 4). Once the event is received the thread is activated in state DD_HANDLE_DATA (line 7) where the data cache entry is set. The second thread, *removeDuplicateData*, issues a similar **when_tuple** request (in line 3), but specifies a different filter function, *filter_DataDuplicate*. That function (not shown) evaluates the same predicate as *filter_data* used by the previous thread and-ed with a check whether the attributes of the new data item are present in the cache. Thus, the action in state RM_DATA_EVENT is

```
/*
_* Store received data items in the data cache
_* (for loop prevention)
_*/
1  rule handleDataItems 30
2  <- exists { data }
3  // Store data item, preventing loops.
4  -> define "dataCache" [source = (data source),
   timestamp = (data timestamp),
   type =(data type)]
5  ....
```

```
/*
_* Drop looping data items.
_*/
1  rule removeDuplicateDataItems 60
2  <- exists { data }
3  <- exists { "dataCache"
   <- eval ( (dataCache source) == (data source) )
   <- eval ( (dataCache timestamp) == (data timestamp))
   <- eval ( (dataCache type) == (data type) )
   }
4  -> retract data
```

Table 3: FACT code for Duplicate Data

```
/*
_* Store received data items in the data cache
_*/
   define tuple data, dataCache [int source, long timestamp, ...]
```

```
1  fsm handleData 30 (byte *context) {
2  define context data;
3  state DD_INIT:
4     when_tuple (data, filter_data,
        DD_HANDLE_DATA);
5     release;
6  state DD_HANDLE_DATA:
7     setNewTuple( dataCache, 1, "%w%w%w", data.source, ... )
   ....
   }
```

```
/*
_* Drop looping data items.
_*/
1  fsm removeDuplicateData 60 (byte *context) {
2  state RM_INIT:
3     when_tuple (data, filter_DataDuplicate,
        RM_DATA_EVENT );
4     release;
5  state RM_DATA_EVENT:
6     remove(data)
7     proceed RM_INIT;
   }
```

Table 4: PicOS Tuples code for Duplicate Data

[1]Strictly speaking, this interface covers two similar interfaces.

triggered by the arrival of the same kind of data item as expected by the first thread whose attributes additionally mark it as a duplicate. Owing to the fact that the priority of *removeDuplicateData* is higher than that of *handleData*, the activation of *removeDuplicateData* will happen before the first thread is able to act upon the duplicate item.

The above form of superimposition, although simple and often effective, is rather crude, e.g., it limits the behavior of all threads waiting on the same data, even though one can envision a situation when such a limitation would be harmful. For example, the data may come in a packet containing some administrative information (metadata) unrelated to the data proper, which some other thread(s) would want to look at. Also, removing duplicates is not all there is: the semantics of data interpretation by the application may call for complex patterns of "removal" affecting different threads in different ways. Generally, the **block_tuple** (illustrated in the next example) is more flexible in this regard.

6.2 Snapshot Debugging

We show how the superimposition principle can be applied to the data handling pattern known as *distributed detection* [7]. The example implements a debug protocol that collects a snapshot of node states from the 1-hop (immediate) neighborhood of the sniffer node [15]. In particular, the snapshot protocol demonstrates the usage of our new **block_tuple** construct.

Consider the code listed in Table 1 describing a fragment of activities of a sensor node, which waits for (and replies to) temperature readout requests. When a request is received, the FSM moves to state S_REPLY and sets a random delay (line 10) before proceeding to state S_TX and sending the response. The role of the delay may be to statistically spread the reports of multiple sensors (which may be expecting the same kind of request) in time as to reduce the number of collisions. We want to slightly modify the code in Table 1 by adding this line:

8.5 SNAP(temperature, temp_sample);

after line 8. This operation adds the temperature tuple to a snapshot cache tuple which collects node states.

```
/*
_* Debug Fsm - Block all other snap threads.
_*/
1  fsm debug 60 (byte *context) {
2  define context  dynamic snapCache
3  state (D_WORK):
4     when_tuple ( DEBUG_TPLID, filter_func(), D_STOP );
5     release;
6  state (D_STOP):
7     block_tuple (all snapCache, block_true_func());
8     send_tuple (all snapCache);
9     when_tuple (DEBUG_RESUME, filter_func(), D_WORK );
10    release;
```

Table 5: Snap-shot debug code

The code of the debug tool is detailed in Table 5. The *debug* FSM waits for a DEBUG message which can be sent by a sniffer program. Once it receives the message, the FSM proceeds to state D_STOP where it blocks all threads that are pending to use a snapshot (logged) tuple in the cache. In the above example, the temperature tuple is blocked from being sent or snapshot again (line 13 in Table 1). The thread then proceeds to send the snap cache to the sniffer node and waits for a DEBUG_RESUME message (lines 8 and 9 in Table 5). Notice that the block remains on while the thread is waiting in the D_STOP state, and is only removed when the state is changed.

7. DISCUSSION AND RELATED WORK

The examples demonstrated how the PicOS-based programming of WSNs can benefit from the principle of separation of concerns using superimposition. The use of the *snapshot debugging* to implement an operations support system (OSS), emphasizes the minimal invasiveness of the control code.

In the data-centric environment of PicOS-tuples, the tuples represent events. This design model can be viewed as a parallel to "conventional" synchronization over data structures or locks. Thus, the tuple parameters of **block_tuple** prove suitable for synchronization. If further distinction is required regarding the tuple instances, a non-trivial block condition function can be provided as the parameter to **block_tuple**.[2].

Our preliminary experience with this latest versionof PIcOS suggests that applying such a blocking idiom in the kernel, and as part of its run-time data structures, is feasible at a reasonable cost. But we admit there is still a need for discussion of the implementation details.

As described in [14] the notions of superimposition form a theoretical basis of aspect oriented constructs. In [18] it is demonstrated how the superimposition notation can be translated to work as AspectJ generic classes using a two phase preprocessing. The notation is called SuperJ and can define new concerns that can be verified separately in the first phase. These definitions are later translated to a collection of generic parameterized aspects and Java classes that can be 'imposed' on other base codes.

The concepts of Linda [8] have been extensively investigated and implemented in various platforms e.g. [17, 22]. The Agilla [22] middleware was one of the first to use flexible mobile agents that could be injected and spread across nodes to preform application tasks. The Agilla system includes all the standard tuple operations of Linda [8] for the local node – in addition to *reactions*, which make it possible to specify a template matching a particular event that the agent is interested in. The **when_tuple** of PicOS is a comparable operation, since it waits on a certain type of events, effectively preforming a reaction. To the best of our knowledge, the explicit **block** notion of PicOS, as influenced by B-Threads, has not been applied before as a modular coordination notion to control and regulate system reactions in general, and in WSN programming models in particular.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed the regulative flavor of superimposition with two examples and we demonstrated how the PicOS-tuples coordination mechanisms address them. In particular, we showed how the operation **block_tuple** can be applied as a central concept of controlling and blocking other thread activities by referring to their tuple operation events. This constitutes a powerful compositional language construct that might be applied in various configurations.

[2]See *bool* (*block_cond(..)) parameter in Table 2

We wish to extend these preliminary results and try other test cases to evaluate where and how the new coordination abstraction can be used in a precise and verifiable manner. In this respect the work [18] gives some insights how this goal can be accomplished, possibly with several flavors of superimposition. Finally, we believe that the generic form of the FSM threads lends itself naturally to be applied in debug supporting tools. Thus, for example, we have prototyped a variant of B-Threads that can interact with a repository, mimicking the PicOS Tuples code almost without modifications. This could be a basis for a Java based sniffer code in the future.

9. ACKNOWLEDGMENTS

This work has been supported in part by grants and in–kind donations from NSERC, iCORE, IBM, and OlsoNet Commumications.

10. REFERENCES

[1] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor networks. *In OSDI'06*.

[2] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[3] Akhmetshina, E. and Gburzyński, P. and Vizeacoumar, F. PicOS: A Tiny Operating System for Extremely Small Embedded Platforms. Proceedings of ESA'03. 116–122, june 2003.

[4] Nicholas M. Boers and Pawel Gburzynski and Ioanis Nikolaidis and Wladek Olesinski. Developing wireless sensor network applications in a virtual environment. Telecommunication Systems, 45(2-3):165–176, 2010.

[5] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of SenSys'06*, pages 167–180, Boulder, Colorado, USA, 2006.

[6] Dijkstra, E.W and C.S Sholten. Termination Detection for diffusing computations. In *Information Processing Letters*, 11(1), North-Holland, August 1980, 1-4.

[7] L. Bougé and N. Francez. A compositional approach to superimposition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 240–249, New York, NY, USA, 1988. ACM.

[8] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[9] M. Chandy and J. Misra. Parallel Program Design *Addison-Wesley*, 1988.

[10] N. Francez and I. R. Forman. Superimposition for interacting processes. In *Proceedings of CONCUR '90*, pages 230–245, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[11] D. Harel, A. Marron, and G. Weiss. Programming coordinated behavior in java. In *Proceedings of ECOOP'10*, pages 250–274, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15:337–356, April 1993.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01*, pages 327–353, London, UK, 2001. Springer-Verlag.

[14] R. Kurki-Suonio. Action systems in incremental and aspect-oriented modeling. *Distrib. Comput.*, 16(2-3):201–217, 2003.

[15] K. Römer and M. Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *Proceedings of the workshop on Real-world wireless sensor networks*, REALWSN '08, pages 36–40, New York, NY, USA, 2008. ACM.

[16] B. Shimony, I. Nikolaidis, P. Gburzynski, and E. Stroulia. Picos tuples: easing event based programming in tiny pervasive systems. In *Proceedings MOMPES '10*, pages 53–60, New York, NY, USA, 2010. ACM.

[17] P. Costa, L. Mottola, and A.L. Murphy, and G.P. Picco, TeenyLIME: transiently shared tuple space middleware for wireless sensor networks. Proceedings of MidSens'06.

[18] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46:529–541, 2003.

[19] K. Terfloth, G. Wittenburg, and J. H. Schiller. Facts - a rule-based middleware architecture for wireless sensor networks. In *COMSWARE*, 2006.

[20] K. Terfloth and J. H. Schiller. Self-sustained Routing for Event Diffusion in Wireless Sensor Networks. In *RuleML '08*, pages 236–241, Orlando, Florida. Springer-Verlag.

[21] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. *Wirel. Netw.*, 8:153–167, March 2002.

[22] Chien-liang Fok and Gruia-catalin Roman and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications In *ICDCS05*, pages 653–662