From Missions to Systems: Generating Transparently Distributable Programs for Sensor-Oriented Systems

Barry Porter Alan Dearle Simon Dobson
University of St Andrews, UK
{bfp, alan.dearle, simon.dobson}@st-andrews.ac.uk

ABSTRACT

Early Wireless Sensor Networks aimed simply to collect as much data as possible for as long as possible. While this remains true in selected cases, the majority of future sensor network applications will demand much more intelligent use of their resources as networks increase in scale and support multiple applications and users. Specifically, we argue that a computational model is needed in which the ways that data flows through networks, and the ways in which decisions are made based on that data, is transparently distributable and relocatable as requirements evolve. In this paper we present an approach to achieving this using high-level mission specifications from which we can automatically derive transparently distributable programs.

Categories and Subject Descriptors

D.2 [**Software**]: Software engineering; D.1 [**Programming**]: Programming techniques; C.2.4 [**Computer-communication networks**]: Distributed Systems

1. INTRODUCTION

Early wireless sensor networks had the relatively simple aim of collecting as much data as possible for as long as possible, with the majority of research focusing on energy-efficient communication protocols to maximise lifetime. This early vision is now evolving as WSNs increase in scale and complexity, break into new application domains and remain relatively expensive and difficult to deploy.

In particular the distinction between the Internet and sensor network worlds is disappearing with the realisation that sensor networks can and should perform significant processing duties on the data that they serve; in addition the services and taskings of sensor networks are seeing an increasing need to evolve over time as stakeholders come and go and their requirements change. These factors are particularly true of the emerging smart cities domain in which a wide range of services is deployed – in some cases on-demand – from multiple stakeholders into a pervasive city-wide sensor network and those services are integrated with systems outwith the sensor network.

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.
MidSens '12 Montreal, Canada
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

In this paper we argue that a high-level mission programming language is needed for the benefit of non-embedded-programming-experts to enable a much wider range of stakeholders to take advantage of deployed sensor infrastructure. Programs written in such a language should be able to be automatically translated (or 'compiled') into a deployable form. We further propose that this approach be underpinned by a computational model in which the ways that data flows through networks, and the ways in which decisions are made based on that data, is transparently distributable and relocatable as requirements evolve.

The specific contributions of this paper are (i) a mission programming paradigm in which programs are constructed from, and reflective of, deployable components; (ii) a taxonomy of component classes and semantic annotations allowing compilation of missions to transparently distributable deployments; and (iii) the identification of automated deployment and refactoring strategies towards optimality.

Our approach is intended to express the elements of a program that exist both inside and outside of a sensor network, ideally allowing seamless deployment and migration of logic between the two spaces as appropriate.

In the remainder of this paper we first discuss related work in Sec. 2; then in Sec. 3 present the key elements of our approach; and in Sec. 4 offer a summary and outlook.

2. RELATED WORK

In the following we discuss the two main bodies of work that relate to ours: research on higher-level software specification and deployment; and research on software evolution in sensor networks (including inter-node code migration).

The desire for higher-level programming abstractions in WSNs has produced a range of ideas such as TinyDB [11], Regiment [16], Logical Neighbourhoods [15] and Flask [12]. The majority of such work proposes a fixed communication / interaction paradigm that is observed to perform well in ad-hoc networks. By contrast we propose an entire system-building model using a mission-programming paradigm from which distributable programs can be generated, involving both control logic and data flow. Regiment and Flask are the closest works to this model that we are aware of, with Regiment proposing an abstraction based on 'streams' and 'areas' over which functions can be applied, and Flask proposing a data-flow model together with a programmatic wiring language. Both however aim at a more fine-grained specifi-

```
1  until [B.location] == ["B.house.location"] {
2      if [time] is after ["5pm"] { send ["Alert"] to [A] }
3      track [B.location] to [mycloud:personTrack]
4      when [B.location] is in ["disallowed locations"] {
5          send ["Alert"] to [A]
6      }
7  }
8  delete [mycloud:personTrack]
```

Figure 1: An example high-level mission program to track a person's location. The program is constructed from a set of verbs, nouns and functions, each of which map to a distibutable component.

cation of system behaviour than that proposed here, leaning more towards 'compilation' down to code rather than 'composition' of pre-built units; both Regiment and Flask also result in a far more static end-result of their compilation process in comparison to our work.

The low-level issues surrounding the remote updating of software in WSNs have been studied in considerable detail. The majority of work focuses on the delivery mechanisms for updates including early work on TinyOS [7, 18] and more recent work on modular solutions [8, 17] and interpreted approaches [9, 1]. The 'stateful mobile modules' in [19] further demonstrate the potential to migrate functionality to enhance performance. While some of this work is relevant to our aims in a low-level mechanical respect, here we examine higher-level reasoning in terms of when and why to modify software, both within a WSN and outside it.

Finally, a smaller body of work has examined the use of policies and automated adaptation in WSNs: [6] for example examines switching between different radio stacks to balance performance with resilience; [20] and [10] suggest the potential to adapt the parameters of components based on context and resource availability; and [3] proposes localised triggering of scripted maintenance protocols when observing pre-set conditions. While we can draw some inspiration from these works in terms of potential performance metrics they fall short of our aim to migrate and refactor software in a much broader sense to optimise entire distributed programs.

3. APPROACH

3.1 Overview

Our aim for high-level mission specification is to provide a programming methodology that is easy to understand for programmers familiar with popular scripting languages.

Missions are therefore expressed in a notation that is relatively close to natural language as shown in Fig. 1. This demonstrates a program to monitor the location of a person and provide alerts if that person fails to arrive home before an expected time or enters a location considered dangerous. The program also tracks the person's location on a continuous basis to a cloud storage service in order to provide further trace information in case an alert occurs. In Fig. 2, part of the corresponding distributable program – composed of a collection of interacting components – that implements this example mission is shown.

3.1.1 Mission Language

Our mission language is designed to model potentially heterogeneous distributed systems. The key differences to classical local programming languages are (i) mission programs

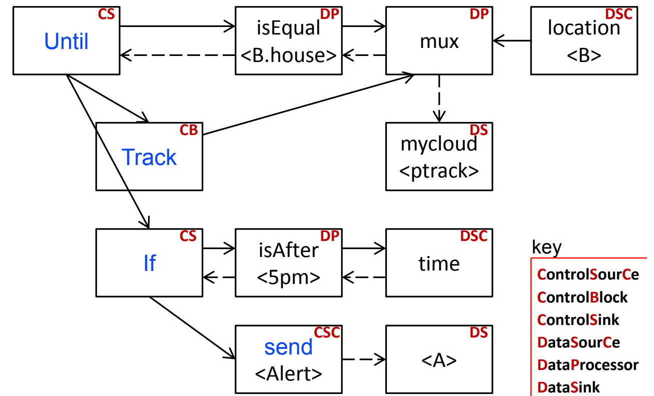

Figure 2: Part of the deployable composition resulting from the mission program in Fig. 1. Solid arrows show control flow and dotted arrows show data flow.

are composed purely of control-flow and data-flow, without 'variables' to hold state; (ii) each individual verb, noun and function in a given expression may map to a *separate implementing component* able to be arbitrarily distributed after compilation; and (iii) the pool of available verbs (and the syntax and semantics of verbs) is extensible via arbitrary specifications in the components that implement those verbs.

In detail, mission program statements are composed of verbs, nouns and functions, where functions are sometimes represented as infix connective terms like '=='. Each 'parameter' of a function is either a noun or another function. Each verb, noun and function is represented at runtime by a corresponding component (discussed below), where components interact via wirings of their interfaces. A parameter to a component representing a function is therefore an input data wire from another component. The syntax of nouns and functions is standardised, while the syntax of verbs is extensible and defined within the implementing component of each verb. Each mission program statement begins with a verb and is followed by nouns / functions arranged in a form indicated by the syntactic meta-description of that verb.

3.1.2 Implementing Components

Key to our approach is that we do not aim to compile the high-level representation down to low-level code, but instead compose programs from a pool of ready-made components.

These ready-made components fall into a fixed taxonomy of component 'classes' with known behaviour and interfaces. Enforcing such a taxonomy of building blocks helps to automate compilation and also aids in general computational reasoning about a system. The taxonomy is based around the need to support both the control flow and the data flow of a distributed program and comprises:

Control Blocks, which are used to unconditionally pass control to other components (in sequence or in parallel); Control Sinks, which conditionally pass control to other control components when events of interest occur from a data component; Control Sources, which produce specific data on a given wire; Data Sources, which produce data such as temperature or location readings; Data Processors, which transform data from one or more input sources to produce one or more output sources; and Data Sinks, which record data to a device. The example components in Fig. 2 are labeled with their taxonomy class.

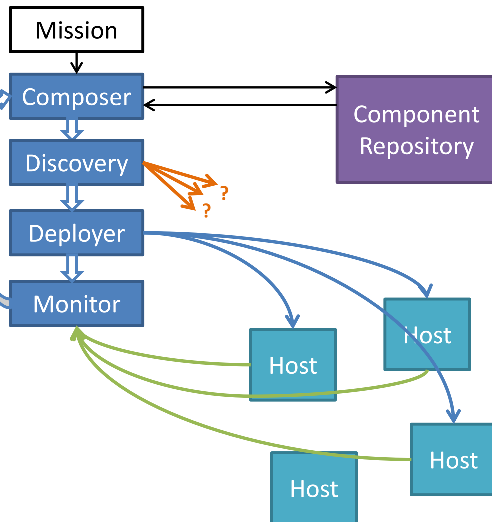

Figure 3: The distributed architecture of our approach relating to the composition, deployment and monitoring of distributed programs. Hosts are expected to include servers and embedded devices.

For control purposes we employ an interface with two operations, *start* and *stop*. The *start* operation passes control to the component and returns when that program branch finishes its execution (or, in the case of infinitely-looping behaviour, may never return); while *stop* forcibly ceases execution of a control component and its associated program branch. For data purposes we employ an interface with the operation *put* which is parameterised with a particular type by implementing components. The way in which this taxonomy is used is described in more detail later.

3.1.3 Distributed Architecture

The distributed architecture of our approach is shown in Fig. 3, also showing the stages used to move from a mission program to a deployed system. A mission program is used as input to a Composer, which parses the mission program and makes semantic matches against ready-made implementation components that are stored in a repository. As output, the composer generates a component graph to pass to the discovery system. The discovery system checks for already-deployed components that match any of the requirements given in the component graph, altering the graph to use such components where possible, before passing the component graph on to the deployer.

The deployer then uses this graph, along with a database of known hosts, to deploy remaining components to appropriate hosts (communicating with runtime support systems on those hosts to do so). Some components, particularly Data Sources such as sensors, will have deployment locations that are implied by their role (e.g. data sources to sense the temperature at a given locale must be deployed on sensor nodes around that locale) while other components, including Data Processors and control elements, can be placed in a variety of locations (i.e. all potential deployment locations are equally 'correct' in terms of program specification but some locations may offer better performance than others).

Finally, to inform refactoring, monitoring feedback filters through the host architecture towards the composer. This feedback may either be acted on before it reaches the composer, in cases where nodes have sufficient knowledge and authority to make autonomous decisions, or else may be acted on only when reaching the composer which (potentially with the involvement of human action) has complete knowledge and permissions to make changes as it sees fit.

The execution of a deployed program then proceeds as follows: control is injected into the first 'line' of the program by a distinguished 'Program' Control Block which executes each line sequentially (thus using the *start* operation and waiting for it to complete). In our example, this entry-point is the 'Until' Control Sink shown in Fig. 2. Control components on which the *start()* operation is used may start additional components including initiating data flow from Data Sources. Control and data flow thus branches through the distributed program as selected components are activated and de-activated following the control logic of the mission.

The above approach has two key benefits: firstly it supports a mapping from high-level language down to a distributable implementation that is sufficiently simple to be machine-automated; and secondly it supports transparent distribution of elemental program components such that each component need not be aware of the physical location of the other components with which it interacts (i.e., they may be on the same host or on different hosts). The latter benefit in particular suggests that we can optimise computation by (for example) moving processing closer to data sources. When actuation is involved we can also move control logic closer to the data sources on which control decisions are based. This, in turn, suggests that we can minimise the amount of data that travels through networks and in sensor networks in particular we can therefore minimise energy consumption.

There are three main problems in realising this approach: (i) the initial compilation procedure to derive an equivalent distributed system matching the mission; (ii) the deployment of these components to appropriate hosts; and (iii) the post-deployment optimisation of the distributed system to adapt to real-time observations. We discuss our work in each of these areas in the following sections.

3.2 Compilation/Composition

A mission program is compiled into an equivalent component graph, the components of which can then be distributed to arbitrary locations (with inter-component connections that span networks transparently proxied through appropriate technologies as required). This procedure is in fact a mixture of compilation and composition – and we generally use the terms interchangeably in this paper – as our approach is not to compile a high-level specification all the way down to machine code, but rather to select from a pool of ready-made components to fit each part of the mission program. The structure of a distributed system is then maintained post-deployment such that it is amenable to refactoring in service of optimisation or fault-tolerance. In the following sections, the meta-annotations that are used on components to support the compilation process are described, followed by the compilation process itself.

Terms used in a mission program are directly mapped to components in our taxonomy. Examining our example program more closely, each statement is comprised of verbs, nouns and functions. A statement must begin with a verb and be followed by one or more nouns / functions and syntactic particles as indicated by the syntactic definition of the particular verb. In Fig. 1, verbs are highlighted in bold;

nouns are provided in square brackets, and functions appear as plain text (in this example all functions are of infix form).

The basis of our compilation process is then that verbs map to Control components; nouns map to Data Sources or Data Sinks; and functions map to Data Processors.

Building on these simple rules, additional meta-structural information and semantic data associated with each concrete component is used to complete the process.

3.2.1 Annotations

Components are annotated in the following ways to aid with the compilation/composition process:

Category and meta-structure.

The mission language particle (verb, noun or function) that a component implements is provided as meta-data along with the component's local structure. This includes the set of required and provided control and data interfaces along with the concrete data types of each data interface.

Name and syntax.

All components provide the name of the verb / noun / function that they implement, and thus the name that can be used to select that component (or more precisely, the set of potential components with this name) within the mission program. In the case of a component implementing a noun or function (i.e. a data component), this information in addition to the component's meta-structure description is sufficient to provide the remaining information using standard syntax. In the case of a component implementing a verb (i.e. a control component), a syntactic specification is also provided as meta-data which contains a **first** symbol that is used to drive the parsing procedure and is followed by terminal and non-terminal symbols describing the syntax of the component. The non-terminal symbols refer (by name) to the component's input data and output control interfaces thus tying structure to syntax. This effectively enables extensible syntax for verb implementations to describe elements such as loops and branches as required.

Semantics.

Finally, the semantics of a component – a description of its abstract behaviour – are provided as meta-data to aid the mission programmer in understanding the way in which a particular component works. While not strictly required for the purposes of compilation, this information is intended to assist the programmer in resolving potential ambiguity. How to describe the semantics of such systems in a meaningful way is an open question. On the one hand we would like something more tractable than structured comments, and richer than normal type signatures. A language such as Promela [14] offers one option here; while process-algebraic approaches offer promise in describing interactions and deriving macroscopic descriptions of global behaviour [2].

Sample components annotated with meta-data in the ways described above are shown in Fig. 4 (with the exception of semantics which are omitted in the interest of brevity).

3.2.2 Compilation

Compilation/composition then proceeds in an automated fashion as follows (using the example in Fig. 1):

The verb **until** is matched to a verb component using the terminal 'until'. If this symbol does not uniquely identify a

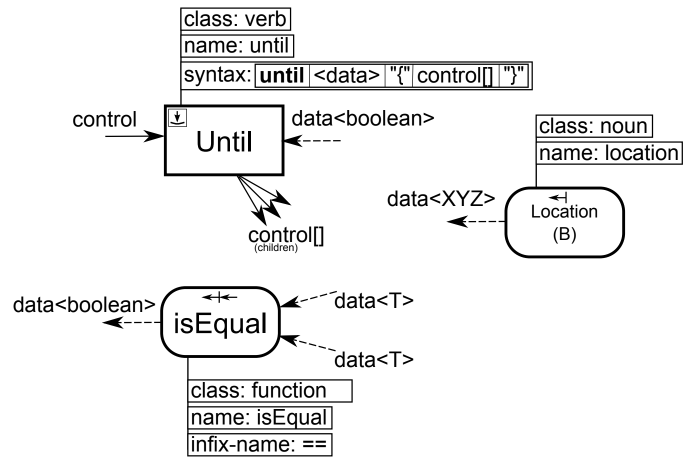

Figure 4: Sample implementing components with concrete meta-annotations.

component, user interaction is solicited to disambiguate the choice of components. Assuming that the desired Until component has been found, its meta-data is queried to establish the number of its interfaces, their type and the component's syntax. An abstract representation of the Until component is shown in Fig. 4 showing that Until has exactly one data input of type boolean and a collection of 0-N output control wires. The syntactic descriptor in the meta-data specifies that the syntax of an Until is:

```
until <data> { control[] }
```

This descriptor is used to drive the parse and match the syntactic structure in the mission language with the component's syntax and structure. The first syntactic child of the component is <**data**> and is specified to be of type data<boolean> in the structural description (see Fig. 4). In the mission program, the statement continues with two nouns (B.location and B.house) and an infix function (equality). The infix function, matched against a Data Processor annotated with the infix-name '==', is first pulled out of the expression and converted to a standard function call with two parameters. The output of this function is taken as the input to the Until component to satisfy its data<boolean> input. The two Data Sources then act as parameters to the isEqual Data Processor (which on a structural level maps to satisfying the data input wirings of isEqual). Type-checks verify compatibility of interactions between selected components, for example that the isEqual Data Processor satisfies the Until component's requirement for a (stream of) boolean value(s). Following the opening '{' the verb components comprising the control outputs of the Until component are matched in the same manner as the Until and this process continues until the closing '}' is parsed.

3.3 Deployment and Refactoring

After the above phase is complete, a distributed program is ready for deployment to appropriate hosts, which may include servers, sensor nodes, and other devices of interest. Matching components to hosts is an NP-hard problem (see e.g. [13]) the details of which we do not examine here. For the purposes of this paper we simply assume that a mixture of heuristics and historical data will provide sufficient guidance for an initial 'rough' deployment plan. After initial deployment, using the component-based model outlined

above, we argue that real-time observations can then be used to hone the deployment plan or refactor the distributed program towards an optimal solution. In the remainder of this section we discuss potential refactoring solutions and the metrics that may inform them.

Note we are interested primarily in *automated* refactoring solutions that can be made without human intervention. We also currently focus on structural refactoring rather than parametric adaptation, though ultimately a mixture of both is likely to be necessary (we leave this to future work). If our mission program composer is viewed as a compiler, the goal of structural refactoring can be likened to the goal of compiler optimisation: to boost the performance of a program without changing its logic. The difference is that we aim to optimise in response to real-time, real-world observations.

Under our approach the components that are most eligible for refactoring are Data Processors, Control Blocks, Control Sinks and Control Sources. Both Data Source and Data Sink components by contrast tend to have more static locations/configurations that are implied by their function.

Migration.

Migration of a component to a different host is the simplest form of refactoring (identified e.g. in [19]). In simple terms, by moving a component whose data input rate is higher than its data output rate (or vice-versa) we can increase performance by decreasing the distance that the larger volume of data has to travel. Migration may also be triggered by a critical resource drain on a component's current host, for example reaching a state in which energy is about to be depleted, or by a state in which a component's current host becomes isolated from the network (in which case the migration will need to be triggered by other hosts). The migration (or not) of a component to a given host may additionally be based on non-real-time metrics such as the historical reliability of that host.

As an example, consider an actuation controller (such as those for HVAC or power distribution [4]), deployed on a server, which makes decisions about when to perform actuations within a WSAN based on data received from that same WSAN. Because this component will likely have a relatively high data input rate in comparison to its data output rate it is reasonable to migrate it from its server position into the WSAN itself. This has the additional benefit that the WSAN gateway nodes are not critical points of failure since the control logic is performed within the WSAN.

Split and Join.

A data processor that is aggregating data from multiple sources to produce a computed output over that data is not always available for simple migration because eventually it will arrive at an equi-distant point from all of its data input sources (or else will oscillate between positions). In this case, to gain further performance benefits in network traffic reduction, it is necessary to split the processor into two or more sub-elements that can migrate down different paths towards the data sources and so provide pre-processing on the input data to reduce the overall traffic level. As an example, consider a component that is computing the average temperature of an entire deployment. It is clearly not possible to migrate such a component close to each of its data sources, so we must split the component into sub-tasks which can migrate towards different regions of the network to perform cluster-based aggregation, computing over locally-available data before sending smaller result packages to the overall averager. A range of similar aggregation approaches may be applied, as surveyed in [5], though care must of course be taken in all aggregation techniques to assure that the functional logic of the original mission program (in this case the fidelity of the data on which it relies) is not compromised.

Load-division.

If a component C is overloaded by input data, and cannot be split into migrateable sub-units (e.g. because there are no available hosts closer to any of the data sources), a further strategy that can be employed is one of load-division in which C is replicated onto a nearby host and an input-splitter is inserted between the data sources and the replicas of C to perform load-balancing (according to a particular algorithm). This strategy is valid for all components whose output, or state machine, does not depend on the entirety of available input data. Examples here include Control Sink components whose sub-behaviours are idempotent or request/response-based (such that action taken pertains to a specific input); Data Processor components that are able to perform valid computation on partial data sets for pipelined processing to further components; or simple Data Sink components that write input data to disk.

Cross-program optimisation.

Finally, when multiple mission programs are distributed into the same network – as will increasingly be the case in infrastructural WSANs – cross-program optimisations are likely to be identifiable by the overall system composers. Common behaviour that is being replicated can thus be shared to the mutual benefit of all deployed systems.

The above collection of strategies represent the kinds of reasoning that – it is hypothesised – it is possible to machine-automate using relatively simple rule-sets without fear of compromising the functional logic of deployed systems. It is envisioned for example that additional meta-annotations on components can be used to understand the ways in which each component can potentially be split, migrated or load-balanced. By applying these kinds of runtime optimisations based on real-time observations, the initial problem of deployment becomes less important in its precision because the system can later optimise autonomously.

4. SUMMARY AND OUTLOOK

This paper presents an approach to generating transparently distributable programs from high-level missions. Each statement in a mission is composed of verbs, nouns and functions which, with the help of meta-annotations, can be used by a compiler to directly map a mission program to implementing components using a taxonomy built around the notions of control flow and data flow.

We have demonstrated the potential to automatically compose such programs and have discussed the ways in which they may be distributed and later refactored taking into account real-time observations. Our approach is novel firstly in its use of extensible high-level language that is inherently mappable to a component implementation, and secondly in the maintenance of system structure post-deployment to support online refactoring towards optimality.

Our current prototypes consist of collections of implementing components (with hand-crafted compositions and deployment plans) that demonstrate the end result of mission-to-system generation, including support for transparent proxying over networks where inter-component wirings span hosts.

In future work we intend to investigate in more detail each of the key problem spaces identified, with particular focus on fully automating compilation and autonomous software refactoring in a variety of scenarios beginning with simple collection and actuation examples. We are also working on developing a concept of 'envelopes' for missions, which will augment the functional logic of a mission program described here with non-functional requirements for concerns such as timeliness and data fidelity. This information will in turn be used further guide the autonomous refactoring strategies that are applied to the deployed system.

5. REFERENCES

[1] N. Brouwers, P. Corke, and K. Langendoen. Darjeeling, a java compatible virtual machine for microcontrollers. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 18–23, New York, NY, USA, 2008. ACM.

[2] M. Calder, S. Gilmore, and J. Hillston. Automatically deriving ODEs from process algebra models of signalling pathways. In *Proceedings of Computational Methods in Systems Biology (CMSB 2005)*, pages 204–215, 2005.

[3] Q. Cao and J. A. Stankovic. An in-field-maintenance framework for wireless sensor networks. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*, pages 457–468, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 197–210, New York, NY, USA, 2010. ACM.

[5] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *Wireless Communications, IEEE*, 14(2):70–87, april 2007.

[6] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. Beven. An experiment with reflective middleware to support grid-based flood monitoring. *Concurrency and Computation: Practice and Experience*, 20(11):1303–1316, 2008.

[7] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM, 2004.

[8] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, New York, NY, USA, 2009. ACM.

[9] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Operating Systems Review*, 36(5):85–95, 2002.

[10] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 211–224, New York, NY, USA, 2008. ACM.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, Mar. 2005.

[12] G. Mainland, G. Morrisett, M. Welsh, and R. Newton. Sensor network programming with flask. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 385–386, New York, NY, USA, 2007. ACM.

[13] D. Menasce and E. Casalicchio. A framework for resource allocation in grid computing. In *Proceedings of 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (IEEE MASCOTS 2004)*, pages 259 – 267, oct. 2004.

[14] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, WIFT '98, pages 90–101, Washington, DC, USA, 1998. IEEE Computer Society.

[15] L. Mottola and G. P. Picco. Programming wireless sensor networks with logical neighborhoods: a road tunnel use case. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 393–394, New York, NY, USA, 2007. ACM.

[16] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, DMSN '04, pages 78–87, New York, NY, USA, 2004. ACM.

[17] B. Porter, U. Roedig, and G. Coulson. Type-safe updating for modular WSN software. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS '11.

[18] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *2nd ACM International Conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[19] M. Strübe, R. Kapitza, K. Stengel, M. Daum, and F. Dressler. Stateful Mobile Modules for Sensor Networks. In *6th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2010)*, volume LNCS 6131, pages 63–76, Santa Barbara, CA, June 2010. Springer.

[20] A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen. A self-adaptive context processing framework for wireless sensor networks. In *MidSens '08: Proceedings of the 3rd international workshop on Middleware for sensor networks*, pages 7–12, New York, NY, USA, 2008. ACM.