Fine-Grained Tailoring of Component Behaviour for Embedded Systems

Nelson Matthys, Danny Hughes, Sam Michiels,
Christophe Huygens, and Wouter Joosen

IBBT-DistriNet, Department of Computer Science,
Katholieke Universiteit Leuven, B-3001, Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. The application of run-time reconfigurable component models to networked embedded systems has a number of significant advantages such as encouraging software reuse, adaptation to dynamic environmental conditions and management of changing application demands. However, reconfiguration at the granularity of components is inherently heavy-weight and thus costly in embedded scenarios. This paper argues that in some cases component-based reconfiguration imposes an unnecessary overhead and that more fine-grained support for the tailoring of component functionality is required. This paper advocates for a high-level policy-based approach to tailoring component functionality. To that end, we introduce a lightweight framework that supports fine-grained adaptation of component functionality based upon high-level policy specifications. We have realized and evaluated a prototype of this framework for the LooCI component model.

1 Introduction

Run-time reconfigurable component models provide an attractive programming model for Wireless Sensor Networks (WSN). As WSN environments are typically highly dynamic, run-time reconfigurable component models allow this dynamism to be effectively managed through the deployment of new functionality or the modification of existing compositions. WSNs are also increasingly expected to support multiple applications in the long-term perspective. In response, reconfigurable component models allow system functionality to evolve to meet changing application requirements. Run-time reconfigurable component models also promote reuse, which is essential in resource-constrained WSN environments.

A number of run-time reconfigurable component models have been developed for embedded systems, most notably OpenCOM [4], RUNES [3], and OSGi [13]. These component models address the problems of dynamism, evolution and reuse by offering developers:

- *Concrete interfaces* that promote the reuse of components between applications.
- *On demand component deployment* that can be used to manage dynamism and evolution through the injection of new functionality.

– *Component rewiring* that can be used to modify component compositions on the fly and thus offers a mechanism to manage dynamism and evolution. The ability to dynamically wire a third party component into a composition also promotes reuse.

In sum, run-time reconfigurable component models allow for reconfiguration of system functionality through the introduction of new components, or the modification of relationships between existing components. However, component-based reconfiguration has two critical disadvantages:

– *Coarse granularity:* As reconfigurations may be enacted only by modifying relationships between components or deploying new components, component-based reconfiguration is a poor fit for enacting fine-grained changes. Thus, while component-based reconfiguration provides a generic mechanism for enacting changes, it is inefficient when that change may be represented by a few lines of code. This is particularly critical for embedded platforms, such as WSN nodes, where memory is limited and software updates are costly operations.
– *Complexity of abstraction level:* Component-based reconfiguration is complex and requires a domain expert to be enacted properly. This complexity prevents end-users from tailoring the functionality of the deployed system themselves. Furthermore, expressing simple changes in a component-based system should be offered at the abstraction level of the end-user.

This paper addresses the problems of coarse granularity and complexity through the introduction of a lightweight policy framework for adapting component behaviour. Policies for this framework are high-level and platform independent, thus allowing end-users to more easily tailor component behaviour. The performance of this system is evaluated through a number of case studies.

The remainder of this paper is structured as follows: Section 2 provides background on component and policy frameworks for networked embedded systems, while Section 3 presents the design of a policy language and corresponding framework for tailoring component behaviour. An initial prototype of this framework is evaluated based on a case study in Section 4. Section 5 critically discusses advantages and shortcomings of our approach. Finally, Section 6 concludes and presents directions for future work.

2 Background

This section firstly discusses the state-of-the-art in component models for networked embedded systems. Section 2.2 then discusses existing policy-based mechanisms for tailoring component functionality. Finally, Section 2.3 provides a brief overview of the LooCI component model.

2.1 Component Models for Networked Embedded Systems

NesC [6] is perhaps the best known component model for networked embedded systems and is used to implement the TinyOS [12] operating system. NesC provides an event-driven programming approach together with a static component model. NesC components cannot be dynamically reconfigured, however, the static approach of NesC allows for whole-program analysis and optimization.

Maté [11] extends NesC and provides a framework to build application-specific virtual machines. As applications are composed using specific virtual machine instructions, they can be represented concisely, which saves power that would otherwise be consumed due to transmitting software modules. However, compared to component-based approaches, Maté has one critical shortcoming - compositions are limited by the functionality that is already deployed on each node and thus it is not possible to inject new functionality into a Maté application without reflashing each node.

OpenCOM [4] is a general purpose, run-time reconfigurable component model and while it is not specifically targeted at networked embedded systems, it has been deployed in a number of WSN scenarios [7]. OpenCOM supports dynamic reconfiguration via a compact runtime kernel. Reconfiguration in OpenCOM is coarse-grained, being achieved through the deployment of new components and modifying connections between components.

The RUNES [3] component model brings OpenCOM functionality to more embedded devices. Along with a smaller footprint, RUNES adds a number of introspection API calls to the OpenCOM kernel. Like OpenCOM, RUNES allows for only coarse-grained component-based reconfiguration.

The OSGi component model [13] targets powerful embedded devices along with desktop and enterprise computers. OSGi provides a secure execution environment, support for run-time reconfiguration and life-cycle management. Unfortunately, while OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [15] consumes more than 80KB, making it unsuitable for highly resource-constrained devices.

2.2 Policy Techniques for Tailoring Component Behaviour

Over the last decade, research on policy-based management [2] has primarily been applied to facilitate management tasks, such as component configuration, security, or Quality of Service in large-scale distributed systems. Policy-based management allows the specification of requirements about the intended behaviour of a managed system using a high-level policy language, which are then automatically enforced in the system. Furthermore, policies can be changed dynamically without having to modify the underlying implementation or requiring the consent or cooperation of the components being governed.

ESCAPE [16] is a component-based policy framework for programming sensor network applications using TinyOS [12]. Similar to our approach, ESCAPE advocates the use of policy rules to govern component behaviour. However, policies in ESCAPE are exclusively used to specify interactions between components,

removing interaction code from the individual components, whereas in our approach we apply policy techniques to configure entire component compositions, including the existing information flow. In addition, ESCAPE is implemented on top of the static NesC component model [6], whereas our policy framework builds on top of a more flexible run-time reconfigurable component model.

Recently, the Service Component Architecture (SCA) defined a Policy framework specification [14], which aims to use policies for describing capabilities and constraints that can be applied to service components or to the interactions between different service components. While not being bound to a specific implementation technology, the SCA policy framework focusses on service-oriented environments such as OSGi [13] which may only be applied to relatively powerful embedded devices.

The approach this paper proposes is to combine the key benefits of a run-time reconfigurable component model (i.e. the ability to inject new functionality dynamically and reason about distributed relationships between components), with the efficiency of policy-based tailoring of functionality. As we will show in Section 4, this reduces the burden on developers while also reducing performance overhead for simple reconfigurations. Furthermore, the policy language we propose is high-level and easy to understand, allowing end-users, as well as domain experts, to customize the functionality of component compositions.

2.3 LooCI: The Loosely-coupled Component Infrastructure

The Loosely-coupled Component Infrastructure (LooCI) [8] is designed to support Java ME CLDC 1.1 platforms such as the Sun SPOT [17]. LooCI is comprised of a component model, a simple yet extensible networking framework and a common event bus abstraction. LooCI components support run-time reconfiguration, interface definitions, introspection and support for the rewiring of bindings. LooCI offers support for two component types, *macrocomponents* and *microcomponents*.

Macrocomponents are coarse-grained and service-like, building upon the notion of Isolates inherent in embedded Java Virtual Machines such as Sentilla [1] or SQUAWK [18]. Isolates are process-like units of encapsulation and provide varying levels of control over their execution (exactly what is provided depends on the specific JVM). LooCI standardizes and extends the functionality offered by Isolates. Each macrocomponent runs in a separate Isolate and communicates with the runtime middleware via Inter Isolate RPC (IIRPC), which is offered by the underlying system. Unlike microcomponents, macrocomponents may use multiple threads and utility libraries.

Microcomponents are fine-grained and self-contained. All microcomponents run in the master Isolate alongside the LooCI runtime. Unlike macrocomponents, microcomponents must be single threaded and self-contained, using no utility libraries. Aside from these restrictions, microcomponents offer identical functionality to macrocomponents in a smaller memory footprint.

Unlike OpenCOM or RUNES, LooCI components are indirectly bound over a lightweight event bus. LooCI components define their provided interfaces as

the set of LooCI events that they publish. The receptacles of a LooCI component are similarly defined as the events to which they subscribe. As bindings are indirect, they may be modified in a manner that is transparent to the composition. Furthermore, as all events are part of a globally specified event hierarchy, it becomes easier to understand and modify data flows.

3 A Policy-based Component Tailoring Framework

3.1 Policy Language Design and Tool Support

The specification of policies to tailor component behaviour is accomplished by using policy rules following Event-Condition-Action (ECA) semantics, which correspond well to the event-driven nature of the target embedded platforms. An ECA policy consists of a description of the *triggering events*, an optional *condition* which is a logical expression typically referring to external system aspects, and a list of *actions* to be enforced in response. In addition, our prototype policy language allows various functions to be called inside the condition and action parts of a policy. By using these policies, we offer a simple, yet powerful method to tailor component behaviour for end-users. In addition, we provide tool support to the end-users to allow simple tailoring of system behaviour. Our tool allows the end-user to firstly select the components and interfaces that can be tailored. Secondly, after specification of the corresponding policies, the tool parses and analyzes each policy for syntactic consistency. Finally, the tool allows the end-user to choose which nodes he wants to deploy the policy to. Concrete examples of the policy language can be found in Section 4.

3.2 Policy Framework Design

As illustrated in Figure 1, the policy framework is deployed on each sensor node and consists of three key components: the *Policy Engine*, the *Rule Manager*, and a *Policy Distribution* component.

The *Policy Engine* is the main component in the framework and is responsible for intercepting events as they pass between two components and evaluating them based upon the set of policy rules on each node. In case of a match (i.e. a *triggering event* and a *condition* evaluating to true), the engine enforces the *actions* defined in the action part of the matching policy. Typical examples of actions are, e.g. denying the event to pass, publishing a custom event, or invoking a particular function in the middleware runtime. Potential conflicts between multiple matching policies are handled by following a priority-based ordering of policies, whereas only the actions of the highest priority policy are executed.

Distribution of policy files from the back-end to the sensor network is achieved using a *Policy Distribution* component hosted on each individual sensor node. After specification and analysis of a policy by our tool, the policy is transformed into a compact binary representation that can be efficiently disseminated to

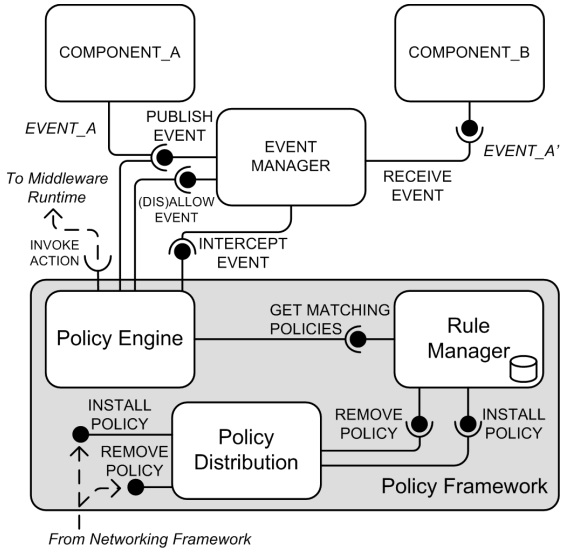

Fig. 1. Overview of the policy framework

the sensor nodes. On reception of this binary policy representation, the policy distribution component passes it to the *Rule Manager* component.

The *Rule Manager* on each individual sensor node is responsible for storing and managing the set of policy rules on the node. After reception of a binary policy from the distribution component, the rule manager converts the policy into a data structure suitable for more efficient evaluation which is then passed to the policy engine on a per triggering-event base. By retaining the ability to dynamically change the set of policies at run-time, the framework can be adapted according to evolving application demands.

4 Case-Study Based Evaluation

This section presents a scenario that requires two archetypal reconfigurations of a distributed component composition: (i.) introduction of filtering functionality and (ii.) binding interception and monitoring. For each case, we compare the overhead of realizing reconfigurations using LooCI macrocomponents and microcomponents to that of realizing reconfiguration using the policy framework introduced in Section 3. Specifically Section 4.1 describes our motivating application scenario. Section 4.2 describes how compositions may be modified through component reconfiguration and policy application. Section 4.3 then considers the overhead for developers inherent in each approach, while Section 4.4 analyzes the memory consumption of each approach. Finally, Section 4.5 explores the performance overhead of component-based versus policy-based reconfiguration.

4.1 Application Scenario

Consider the scenario of a WSN-based warehouse monitoring scenario. In this scenario, a company STORAGO_CO provides temperature controlled storage of goods, wherein the temperature of stored packages is monitored using a WSN running the LooCI middleware. STORAGE_CO offers two classes of service for stored goods: *best effort temperature control* and *assured temperature control*. The customers of STORAGE_CO (CHOCOLATE_CO and CHEMICAL_CO) each have different storage requirements that evolve over time.

- *Best effort temperature control*: in this scheme, STORAGE_CO sets temperature alarms, which alert warehouse employees if the temperature of a stored package has breached a specified threshold. As the scheme is alarm-based, it generates low levels of traffic, increasing battery life and reducing cost.
- *Assured temperature control*: in this scheme, STORAGE_CO provides continuous data to warehouse employees, who may view detailed temperature data and take pre-emptive action to avoid package spoiling. As this scheme transmits continuous data, it decreases node battery life and increases costs.

Scenario 1: CHOCOLATE_CO begins by requesting the *assured temperature* service level from STORAGE_CO, however, due to tightening cost-constraints, CHOCOLATE_CO later requests their service level to be switched to *best effort*. CHEMICAL_CO begins by requesting the low-cost *best effort* service, however stricter government regulations require CHEMICAL_CO increasing their coverage to *assured temperature control*.

Scenario 2: STORAGE_CO wishes to perform a detailed analysis of how their WSN infrastructure is being used, and thus deploys functionality to monitor all component bindings in their WSN. This functionality includes accounting of all events that pass.

4.2 Component-based Modification versus Policy-based Modification

Scenario 1: This section explores how the changing requirements of the customers on both temperature monitoring schemes can be reflected using (i.) component-based modification of the compositions, and (ii.) by a single composition customized using our policy-based approach.

Component-based Tailoring of Functionality: The *assured* and *best effort* temperature monitoring schemes discussed in Section 4.1 may be represented by two distinct component compositions. This is shown in Figure 2. In the *assured* monitoring scheme, a TEMP_SENSOR exposes a single interface of type TEMP, which is wired to the matching receptacle of a TEMP_MONITORING component. In the *best effort* temperature monitoring scheme, the TEMP_SENSOR

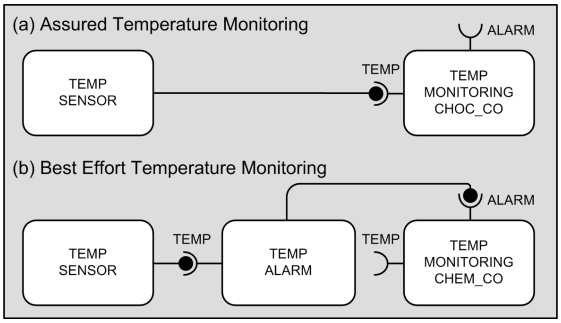

Fig. 2. Component configurations

component is wired to the matching receptacle of a TEMP_ALARM component, the ALARM interface of which is then wired to the matching interface of a TEMP_MONITORING component.

In the case of CHOCOLATE_CO, switching from *assured* to *best effort* temperature monitoring, the existing TEMP_SENSOR component will be unwired from the TEMP_MONITORING component and rewired to a TEMP_ALARM component, the ALARM interface of which is wired to the TEMP_MONITORING component.

In the case of CHEMICAL_CO, switching from *best effort* to *assured* monitoring, the existing TEMP_ALARM component will be unwired from the TEMP_MONITORING and TEMP_SENSOR component. Subsequently, the TEMP interface of the TEMP_SENSOR component will be wired to the matching receptacle of the TEMP_MONITORING component.

Policy-based Modification: To enable CHOCOLATE_CO to switch from *assured* to *best effort* monitoring, the developer needs to specify and enable the following policy with priority 1:

```
policy "assured-to-best-effort" "1" {
  on TEMP as t; //TEMP contains (source,dest,value)
  if(t.value > 20 && t.dest == TEMP_MONITORING_CHOC_CO)
  then( //publish an ALARM event to TEMP_MONITORING_CHOC_CO
   publish ALARM(t.source, TEMP_MONITORING_CHOC_CO, t.value);
   deny t; //and block TEMP event for further dissemination
  )
}
```

This policy specifies that the policy engine should intercept all TEMP events, while only allowing those events to pass with a temperature value higher then 20 degrees Celsius and by converting them to ALARM events destined for the TEMP_MONITORING component.

To enable CHEMICAL_CO switching from *best effort* to *assured* temperature monitoring, the developer needs to specify and enable the following policy:

```
policy "best-effort-to-assured" "1" {
   on TEMP as t;
   if(t.dest == TEMP_ALARM)
   then( //allow sending to TEMP_ALARM for threshold checking
      allow t;
      t.dest = TEMP_MONITORING_CHEM_CO; //change destination
      publish t; //assure sending to TEMP_MONITORING_CHEM_CO
   )
}
```

This policy changes the destination of TEMP events from the TEMP_ALARM to the TEMP_MONITORING_CHEM_CO component to enforce the assured monitoring scheme. In addition, it may not break the existing composition (i.e. TEMP events must also be sent to the TEMP_ALARM component).

Scenario 2: Insertion of Global Monitoring Behaviour: The network-wide monitoring of component interactions described in Section 4.1 may also be implemented using a component-based or policy-based approach. In either case, the reception and transmission of an event should be logged to a ACCOUNTING component which stores events for future retrieval and analysis. In order to implement logging or accounting using component-based modification, STORAGE_CO would be required to continually probe the network to discover the state of compositions and then insert a BINDING_MONITOR interception component into each discovered binding - clearly a resource intensive process.

In contrast, as the LooCI Event manager provides a common point of interception for all events on each node, a single, generic policy may be inserted to perform equivalent monitoring. As all events are routed through the policy engine, such a configuration is agnostic to the component compositions executing on the WSN, and clearly entails significantly lower overhead. A policy to implement this is shown below:

```
policy "logging" "1" {
 on * as e; //all events have source, dest, data[] as payload
 then( //always do accounting of event occurrence
   invoke ACCOUNTING(e.source, e.dest, e.data[]);
   allow e; //do not block e, allow it to continue
 )
}
```

While this example is simple, we believe that the ability to install per-node, as well as per binding policies to enforce various non-functional concerns may reduce overhead in many scenarios.

4.3 Overhead for the Developer

In this section, we analyze the effort required to implement the TEMP_ALARM component and compare this with the effort required to develop a functionally

Table 1. Development Effort Comparison

Micro-component	Macro-component	Policy
35 SLoC	35 SLoC	8 SLoC

equivalent policy, as described in Section 4.2. Each implementation was analyzed in terms of Source Lines of Code (SLoC). The results are shown in Table 1.

Perhaps more critically than the conservation of development effort, as illustrated by the SLoC savings shown in Table 1, is the high-level and platform independent nature of the policy specification language, which unlike a Java-based LooCI component could equally be applied to a TinyOS [12] or Contiki [5] software configuration where a suitable policy interpreter exists.

4.4 Memory Footprint

The size of the policy framework is 26 kB. Subsequently, we analyzed the static memory (size on disk) and dynamic memory (RAM) consumed by the software elements introduced in Section 4.2. As can be seen in Table 2, policy-based reconfiguration consumes significantly less memory than component-based reconfiguration, a critical advantage in memory-constrained environments like WSNs.

Table 2. Memory Consumption

	Micro-component	Macro-component	Policy
Static	1 kB	1 kB	103 bytes
Dynamic	3 kB	26 kB	376 bytes

4.5 Performance Overhead

We evaluated the performance of policy-based and component-based reconfiguration using a standard SunSPOT node (180 MHz ARM9 CPU, 512 kB RAM, SQUAWK VM 'BLUE' version) and a 3 GHz Pentium 4 desktop with 1 GB of RAM running Linux 2.6 and Java 1.6. We first logged the time required to deploy and initialize the policy specification and component implementation required to achieve the reconfigurations described in Section 4.2. We then analyzed the time which each took to handle an incoming TEMP event (i.e. process it and disseminate an ALARM event to the gateway). In each case, the SPOT node was deployed between 20 cm and 30 cm from the network gateway and we performed 50 experiments, the averaged results of which are illustrated in Table 3.

As can be seen from Table 3, not only is the overhead inherent in deploying and initializing a policy significantly lower than that of deploying and initializing a component, the ongoing performance overhead per event caused by applying a policy to a binding is also lower (or equal to microcomponent performance) than that caused by inserting a new macrocomponent. In embedded environments where CPU and energy resources are scarce, we believe that policy-based reconfiguration provides concrete benefits over component-based reconfiguration for tailoring compositions as it does not introduce additional overhead.

Table 3. Performance Comparison

	Microcomponent	**Macrocomponent**	**Policy**
Deployment	11330 ms	11353 ms	200 ms
Initialization	8418 ms	7420 ms	6 ms
Execution overhead	28 ms	43 ms	28 ms

5 Discussion

The evaluation presented in the previous section clearly shows that policy-based modification of component compositions can have significant advantages in terms of: (i.) lowering development overhead, (ii.) reducing memory footprint and (iii.) improving performance. This leads to a critical question: When to apply component-based modification of functionality, and when to use policy-based tailoring of functionality?

The policy-based approach is suited to enforce non-functional concerns like accounting or security on component compositions, as these non-functionalities are orthogonal to the composition and not radically change the end-to-end information flow in the component composition.

Despite the concrete advantages of policy-based composition modification, this approach is not without drawbacks: it can reduce reusability of components. In a pure (or functional) component composition, the functionality of each component is solely identified by its type along with the interfaces and receptacles it provides. As the application of policies to component bindings can modify functionality in a manner that is opaque, this can effectively render the component unreliable for use in other compositions and thus reduces the maintainability of the system.

Managing long-term system evolution must be done with care. Rather, we believe that policies should be used to efficiently realize transient modifications to compositions and to enforce non-functional concerns on compositions.

6 Conclusions

This paper has presented a policy-based framework that can be used to tailor the functionality of component compositions. We have presented a compact and lightweight prototype of this framework realized for the LooCI component model and, through evaluation, we have shown that policy-based tailoring can reduce overhead for developers, reduce memory consumption and improve the performance of reconfiguration when compared to purely component-based reconfiguration approaches.

In the short term, future work will focus upon further researching the impact of policy-based modifications on component compositions. In addition, we plan evaluating policy-based tailoring of functionality in a logistics scenario with concrete WSN end-users. In the longer term we hope to improve the expressiveness of our policy language, and implement prototypes of our policy engine and evaluate its performance for the OpenCOM [4] and OSGi [13] component models.

Acknowledgments: Research for this paper was partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, Research Fund K.U.Leuven, and is conducted in the context of the IBBT-DEUS project [9] and IWT-SBO-STADiUM project No. 80037 [10].

References

1. Sentilla Perk Platform. http://www.sentilla.com/ (July 2009).
2. R. Boutaba and I. Aib. Policy-based management: A historical perspective. *J. Network Syst. Manage.*, 15(4):447–480, 2007.
3. P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2):149–162, 2007.
4. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.
5. A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.
6. D. Gay, P. Levis, R. V. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, USA, May 2003. ACM Press.
7. 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. *Conc. Comp.: Pract. Exper.*, 20(11):1303–1316, 2008.
8. D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. Del Cid, S. Michiels, C. Huygens, and W. Joosen. LooCI: a loosely-coupled component infrastructure for networked embedded systems. Technical Report CW 564, K.U.Leuven, Sept. 2009.
9. IBBT-DEUS project. https://projects.ibbt.be/deus (July 2009).
10. IWT STADiUM project 80037. Software technology for adaptable distributed middleware. http://distrinet.cs.kuleuven.be/projects/stadium/ (July 2009).
11. P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, USA, 2002.
12. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. A. Brewer, and D. E. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 1–14, Mar. 2004.
13. OSGi Alliance. About the OSGi Service Platform, whitepaper, rev. 4.1, June 2007.
14. OSOA. SCA Policy Framework. SCA Version 1.00, March 07.
15. J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. *SIGOPS Oper. Syst. Rev.*, 41(3):245–258, 2007.
16. G. Russello, L. Mostarda, and N. Dulay. Escape: A component-based policy framework for sense and react applications. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 212–229, 2008.
17. Sun Microsystems. Sun SPOT world. http://www.sunspotworld.com/ (July 2009).
18. Sun Squawk Virtual Machine. http://squawk.dev.java.net/ (July 2009).