Streamlining Development for Networked Embedded Systems Using Multiple Paradigms

Christophe Huygens, *Katholieke Universiteit Leuven*

Danny Hughes, *Xi'an Jiaotong-Liverpool University*

Bert Lagaisse and Wouter Joosen, *Katholieke Universiteit Leuven*

In networked embedded systems, multiparadigm programming enables an integrated approach for developing complementary artifacts that are essential but can't be programmed using a single paradigm.

The stringent requirements imposed by mobility and adaptation scenarios amid scarce energy and system resources, as is typical for networked embedded systems, mandate optimization throughout the hardware-software life cycle. Enabling this optimization requires addressing the embedded system's life cycle in an integrated way, avoiding a single-application, development-centric focus. Deployment, operational activities, and maintenance require software development activities that must all be considered to become integral parts of the software development continuum. We can no longer afford the traditional rift between software development, system engineering, and system management. Yet the actors of the continuum are highly diverse regarding life-cycle-phase participation, goals, typical processes, and skills, so an integrated approach must respect this diversity (see the "Key Activities and Paradigms" sidebar).

Multiparadigm programming provides an integrated approach for developing complementary artifacts that are essential in a distributed embedded system's full life cycle but can't be programmed using a single paradigm. This challenging new setting raises several questions regarding the organization of software development. Who are the typical actors in a solution using networked embedded systems? How can we divide the development task among these process actors? And, from a development perspective, what are appropriate programming approaches that align with both the objectives and the actors' skill sets? We've identified, applied, and validated component-based development, policy-driven administration, and constraint-based goal declarations at multiple abstraction levels to address this challenge. In this article, we present some lessons learned regarding this integrated approach, based on our research activities and prototypes developed in cooperation with industry partners.

Actors of the System-Software Continuum

Future networked embedded systems will be federated infrastructures that share resources among

0740-7459/10/$26.00 © 2010 IEEE

Key Activities and Paradigms

The following are the key activities and paradigms for networked embedded-systems development:

- *Traditional software development that delivers business and application logic.* Software developers typically produce communicating software components according to component-based software engineering principles.
- *Development of predeployment artifacts.* Using aspect-oriented composition results in instrumentation of components or their communication by selective insertion of policy engines. These policy engines allow more dynamic management of cross-cutting concerns such as security and optimization.
- *Development of deployment artifacts.* High-level, declarative abstractions describe the various actors' goals and the application compositions, resulting in the assignment of resources, loading and wiring of components, and/or loading of policies and associated code.
- *Development of specific runtime artifacts that can affect application logic at execution time.* Declarative or imperative policy definitions can externally influence the running system's behavior.

In an integrated approach, development isn't an isolated activity at the beginning of the life cycle. Artifacts for various life-cycle phases are produced continually by the actors and incorporated into the target system at regular intervals.

different applications. The infrastructure owner controls his or her pool of devices, comprising part of the federation. Applications could span multiple administrative boundaries, reflecting the application owner's business goals. Component developers provide prepackaged functionality (ultimately ending up in system repositories) to support the infrastructure and application owners' goals. When functionality is unique to the application, the application owner must develop it.

Life Cycle of a Networked-Embedded-System Application

The application owner composes the application in terms of existing or new components. He or she then submits the composition (and potentially new component code) to the infrastructure owner, along with quality objectives. The infrastructure owner also has goals, usually system-wide requirements driven by concerns such as system lifetime optimization or service levels with various application owners.

The composition and actors' goals are inputs to the *deployment phase*. This phase's objective is to merge all interests, look for a solution that meets a maximum of goals, and translate the goals into actual deployable scenarios. Execution of this scenario then takes place during the *runtime phase*. In this phase, substitution of application elements is possible, and the infrastructure owner should have access to mechanisms that can influence the ensemble of running applications and thus fine-tune system functionality.

Predeployment activities can precede the deployment phase. Through these activities, the infrastructure owner can process and manipulate the components however he or she sees fit. Within a single networked embedded system with multiple applications, some applications might be running at any given time, while others are in the predeployment and deployment phases.

Development Activities and Programming Approaches

The actors' needs in the development process are diverse, as is evident from the divergent approaches that system and software engineers use. To achieve good results, networked-embedded-system developers must be system aware. In addition, their infrastructural counterparts must step up from low-level configuration management to an abstraction level adequate for cross-actor optimization.[1] Table 1 shows the approach we propose.

Development Activities for the Runtime Phase

Given wireless sensor network (WSN) platforms' limited capabilities and emphasis on sense-process-send, most of the functionality needed for sensor applications will already be available in the WSNs (see the "Wireless Sensor Networks" sidebar). A *component-based programming model* offers the embedded-system developer reuse of existing artifacts and a flexible deployment and wiring model, without dictating a specific host language.

Unfortunately, a component-based approach isn't attractive to infrastructure owners. Concerns exist that are difficult for component developers to recognize or that must be addressed independently (for instance, because of frequent change). For example, in a building-management scenario (see the "Hospital Building-Management Case Study" sidebar), we might temporarily want to suspend all components to devote resources exclusively to the fire detection and control component. As an example of a concern that is to be addressed independently, we need to make sure that a patient's insulin pump is never actuated by an HVAC control component. To some extent, we could address these concerns at the composition or deployment level,

Table 1

Development activities (specification by actor of behavior in life-cycle phase) for networked embedded systems

	Actor		
Life-cycle phase	Component developer	Application owner	Infrastructure owner
Predeployment			
Abstraction level	Not applicable	Not applicable	High
Development activity	Not applicable	Not applicable	Declarative aspect-oriented specification of component composition
Deployment			
Abstraction level	Not applicable	High	High
Development activity	Not applicable	■ Declarative constraint-based specification of application qualities ■ Declarative specification of component composition	Declarative constraint-based specification of systemwide qualities
Runtime			
Abstraction level	Low	Low	Mid
Development activity	Component-based development	Component-based development	Declarative or imperative rule-based specification of policies

but the resulting redeployment or rewiring operations are expensive in the networked embedded environment. Offering the infrastructure owner a dynamic mechanism to fine-tune runtime behavior through simple dynamic *declarative or imperative policies* provides a solution. Policy updates are compact in terms of both footprint and code, and can be efficiently implemented and executed. The declarative or imperative nature aligns well with the system administrators' view.[1]

Development Activities for the Predeployment Phase

The middleware can include the policy engine so that all interactions with other components must pass this policy engine. However, this strategy has two key disadvantages. First, the middleware would be bloated with potentially unused functions. Second, pervasive policy evaluations would entail overhead. A more elegant solution is to allow granular inclusion of the policy engine by the infrastructure owner during the predeployment phase—for example, by using *aspect-oriented composition* combined with implementation of the policy engine as a true component. Aspect-oriented composition is also useful for describing concerns that aren't policy driven but that can be implemented by including other components, or

Wireless Sensor Networks

Wireless sensor networks are a class of networked embedded systems with nodes usually consisting of an energy source, a low-power CPU, a limited amount of memory, a radio, and some I/O capability such as a temperature sensor.[1,2] WSNs are characterized by high mobility, minimal capabilities, and low reliability. From a network perspective, the nodes are mostly self-organizing, cooperating to achieve not only network-wide services such as routing and reliable node-to-node communication but also more complex services such as data aggregation, localization, and rogue node detection. These services are all highly optimized for concerns such as energy conservation or security. A sensor network is often deployed in a tiered architecture, ranging from extremely limited devices over cluster heads to network gateways. Such gateways link the sensor network to services in the wired back end, which handle management functions such as code deployment and resource allocation. WSNs introduce additional complexity via *sharing* (many applications want to use a limited amount of resources) or *federation* (nodes having a limited trust relation cooperate toward an overall goal). Because of the environment's highly dynamic nature, the need for high-level abstractions, and the many concerns and actors involved, WSNs are ideal for testing the limits of software engineering frameworks.

References

1. M.-M. Wang et al., "Middleware for Wireless Sensor Networks: A Survey," *J. Computer Science and Technology*, vol. 23, no. 3, 2008, pp. 305–326.
2. J. Yick, B. Mukherjee, and D. Ghosal, "Wireless Sensor Network Survey," *Computer Networks*, vol. 52, no. 12, 2008, pp. 2292–2330.

Hospital Building-Management Case Study

This case study discusses how networked embedded systems can be used in a hospital context, where a mix of wireless and stationary nodes support the many different business processes of the various actors present. Because of the regulatory environment, healthcare scenarios also easily uncover complex requirements, primarily in the security or assurance domain. Many applications run on a hospital's stationary nodes. For example, Figure A illustrates a flu detection ap-

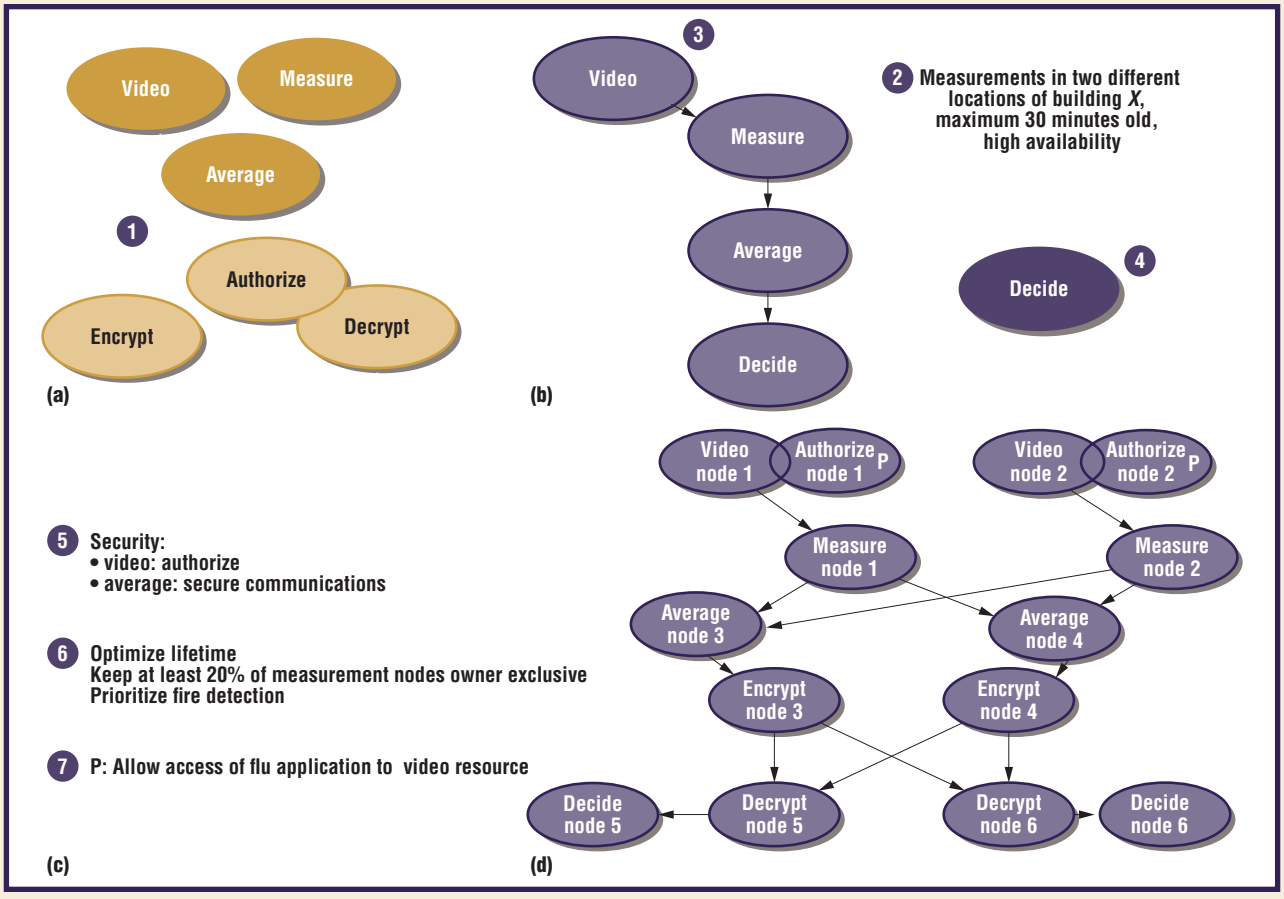


Figure A. Artifacts as produced by (a) component developers, (b) the application owner, (c) the infrastructure owner, and (d) the plan (components, wirings, and policies ready for deployment) with weaving as an integration strategy. Component developer activities include (1) component-based development. Application owner activities include (2) declarative constraint-based specification of application qualities, (3) declarative specification of component composition, and (4) component-based development. Infrastructure owner activities include (5) declarative aspect-oriented specification of component composition, (6) declarative constraint-based specification of systemwide qualities, and (7) declarative or imperative rule-based specification of policies. (P indicates a policy artifact.)

that influence the wiring between components similar to the way deployment specifications do.

Development Activities for the Deployment Phase

Low-level reprogramming of networked embedded systems is impractical due to scale, heterogeneity, and node mobility. Development in this phase involves describing the actors' objectives using high-level abstractions, ultimately resulting in the loading and wiring of components, policies, and their associated code. The application's *quality objectives*, such as desired coverage or data timeliness, are best described declaratively using constraints. The same applies to the system's quality objectives, where concerns such as precedence between applications or overall system lifetime optimization come to mind. Besides

plication. The application owners in this case study include building-management personnel, administrative support personnel, and medical staff. The nodes provide services to those application owners by running applications such as physical access or HVAC control, equipment asset tracking, or even medical functions such as group patient monitoring.

In addition, because of recent developments, the medical application owner wants to deploy a flu-monitoring system. To assess patient temperature, this system reuses the existing infrared capabilities of cameras connected to building nodes for physical security and fire prevention. To build this functionality, the medical application owner composes the application on the basis of the system's existing components, as provided by component developers, such as image-based person detection and temperature estimation. But the medical application owner provides his or her own decision-support system. This new application doesn't need to run in every room; a statistically significant sample is sufficient. The medical application owner then submits the new components, the composition, and the quality goals, such as desired coverage statistics, to the infrastructure owner—in this case, the hospital IT department.

During the predeployment phase, the IT department analyzes the components from various perspectives. For example, respecting security concerns mandates communications security for patient data. Moreover, access to critical components such as image-based person detection must be controlled. The paradigm-integrating middleware then merges the flu-monitoring application's quality objectives (with a minimum coverage of two rooms) with the infrastructure owner's objectives (the fire-control system has absolute priority) and the flu application's composition, yielding an actual deployment of instrumented components and associated wiring. During runtime, it's still possible to change, control, and augment the flu application. Better detection algorithms could become available, resulting in substitution. Or an outage of several nodes could deactivate this long-term, noncritical monitoring function—or even initiate an undeployment action—to free up resources for higher-priority applications.

the application's quality objectives, the application owner must also describe its core business logic. In component programming, composing boils down to a *declarative configuration* of links between the provided and required interfaces of the components involved.

Different Middleware

We can make two observations regarding our approach. First, it doesn't strictly adhere to the traditional sequence of development ⇒ deployment ⇒ operation. Without any component development activity, it's possible to reinitiate the predeployment, deployment, and runtime phases because of simultaneous development activities or system dynamics. Therefore, it's appropriate to consider programming for the complete life-cycle continuum rather than having isolated development silos. Second, the multiparadigm approach requires more than platforms, development tools, and integrated development environments (IDEs). It also requires next-generation middleware to provide an integrated execution environment for the various types of artifacts.

Programming with Dissent Middleware on the Sun SPOT Platform

Here, we present our Dissent (Distrinet Sensor Network Toolkit) middleware as an illustrative case of our approach to embedded development. Table 2 revisits the various development activities in the context of our middleware solution based on the Sun SPOT (Small Programmable Object Technology) platform (www.sunspotworld.com). Sun SPOT is a WSN device based on the Squawk Java Micro Edition (ME) virtual machine.

Runtime Phase

The component model in Dissent is lightweight in terms of both programming abstraction and memory footprint. The developer defines a component by extending a generic component base class, which provides standard component manipulation methods (such as start, stop, and resume). Extending this base class also provides access to a communications framework based on an event bus that promotes loose coupling between cooperating components. Converting a Java ME application into a component requires only a base class modification and an **init()** call.

Components publish and subscribe to events. Component wiring initiates the event flow to a particular component. Events carry all information in the network, whether application data, state information, or rewiring instructions. The code footprint of the middleware underpinning this model is modest for both the component and communications abstractions. Memory consumption and per-

Table 2

Development activities for Dissent (Distrinet Sensor Network Toolkit) middleware on Sun SPOT (Small Programmable Object Technology) sensors

	Actor		
Life-cycle phase	Component developer	Application owner	Infrastructure owner
Predeployment			
Abstraction level	Not applicable	Not applicable	Low
Development activity	Not applicable	Not applicable	Byte code instrumentation of Java Micro Edition (ME) target code using AspectJ
Deployment			
Abstraction level	Not applicable	High	High
Development activity	Not applicable	■ Domain-specific language for coverage qualities ■ Declarative XML directed graphs for compositions	Object-oriented domain modeling and constraint rules (declarative policy)
Runtime			
Abstraction level	Low	Low	Mid
Development activity	Java-based components (extend Java ME MIDlets)	Java-based components (extend Java ME MIDlets)	Event-condition-action rules (imperative policy)

formance overhead also validate feasibility on small platforms.[2] Although using Sun SPOT automatically implies Java and an object-oriented paradigm, these aren't mandatory for our approach. The key runtime paradigm is the component concept and the associated event-based interaction.

To offer the infrastructure owner a suitable runtime abstraction, we use event-condition-action (ECA) rules. This aligns well with the event-based communications model, and a more imperative approach is well established among systems engineers. The middleware planner can generate most ECA policies automatically from the deployment specifications (see Figure 1). But we also facilitate system-owner control by providing support tools for specifying, checking, and deploying policies. An analysis of policy language, code size, resource consumption, and overhead of the corresponding middleware is available elsewhere.[3] The action part of the rules is diverse: events can be manipulated, they can be initiated or redirected to other components, and methods can be called.

Figure 1 demonstrates how to integrate a typical nonfunctional requirement such as channel security without involving the application owner or component developer. As location events are generated and consumed, the security primitives are applied, and the event proceeds to its original destination.

Deployment Phase

Deployment-related activities in our solution mainly use a domain-specific language with support to express constraints that specify qualities. Expressing constraints inevitably involves defining metrics to specify intended behavior. In networked embedded systems, our target is either a node or a group of nodes. For a group, we specify intended coverage. However, given the target's unreliable and dynamic nature, achieving a fully covered, atomic deployment is difficult. Unlike more traditional systems, in networked embedded systems, deployment takes time to propagate, and full coverage might never be achieved. We further detail the deployment using consistency metrics, which express deployment consistency in terms of completeness and timeliness. We've also explored modeling the infrastructural domain using object-oriented techniques,[4] and we've subsequently started applying constraint-based rules to the modeled universe to express goals.

The application owner describes compositions using a directed (ordered) graph of components,

which relates components either from a catalog or as provided by the application owner. The middleware must then enact the declarative deployment definitions, a process involving several semiautomatic steps, resulting in an actual deployable configuration or plan:

1. Some form of precedence, as set by the infrastructure owner, must resolve conflicting objectives. For example, security and performance objectives could conflict.
2. To satisfy each objective, choose a tactic. For example, implement security using a policy mechanism, but obtain data accuracy by duplicating measurement components.
3. Find a solution set and build the associated deployment plan. The deployment plan corresponds to a series of targeted `deploy_component()`, `wire_component()`, or `deploy_policy()` instructions.
4. Reiterate the first three steps at periodic intervals as applications are added, application or system objectives change, or state changes occur in the nodes because of mobility or resource consumption.

Because the WSN architecture's resource-rich back-end tier can handle most of this advanced deployment problem, it's possible to apply complex scheduling and planning algorithms with little performance impact on the networked embedded system.

Predeployment Phase

Our middleware doesn't currently handle aspect-oriented composition declarations. We still need to detail the concerns at a low abstraction level. However, simple template-based translation from declaration to implementation artifacts appears to be realistic for some concerns. To provide for the aspect-oriented instrumentation that the infrastructure owner will use to insert the policy engines, we use AspectJ, merging the Java ME and AspectJ tool chains. Again, because instrumentation takes place in the resource-rich back-end tier, this activity's performance and resource consumption isn't an issue. The AspectJ runtime required for the weaving operation is large, but we can reduce it to a few kilobytes before including it in the deployment unit by deleting all AspectJ language constructs except the exception handlers. Figure 2 shows how the infrastructure owner can intercept specific operations in submitted code, albeit at a very low abstraction level.

An obvious use of such interception is the redirection of resource access to the ECA-driven policy engine. To tackle the problem of tight coupling between the artifacts produced by different actors, we don't apply aspect-oriented programming universally. Instead, we limit its application to functionalities that are well-known to component, application, and infrastructure actors. Examples of such functionalities include SPOT application-independent API calls and component communication over the event bus.

```
policy " enable security of location data " "1" {
   on LOCATION as loc;
   if( true ) // always executed
   then ( // enable confidentiality and integrity
            invoke security . encrypt_payload ( loc);
            invoke security . attach_mac (loc);
            allow loc;
   )}
(a)

policy " check security of location data " "1" {
   on LOCATION as loc;
   if( ( loc . dest == PHARMA\_TRACKER\_COMP ) && security . check_mac (loc ) )
   then ( // decrypt payload and allow
            invoke security . decrypt_payload ( loc);
            allow loc;
   )}
(b)
```

Figure 1. Runtime event-condition-action policy as specified by the infrastructure owner to enforce nonfunctional requirements—in this case, channel security: (a) a policy to apply data security to location data when sending, and (b) a policy to check and remove location data security upon receipt at the destination. The policy enforcement points are integrated with the event bus and thus intercept the components' event interaction mechanism (the events of type loc in the example).

```
package org.sunspotworld;
import com.sun.squawk.*;
public aspect intercept {
   private Isolate isolate; private int iid;
before(ILightSensor ls): target(ls) && call (* getAverageValue(..)) {
      isolate = com.sun.squawk.Isolate.currentIsolate();
      iid= isolate.getId();
      if (!engine.allow("iid"+iid,"lightsensor","get"))
      {throw new IllegalAccessException();}
      }
```

Figure 2. Code snippet illustrating the interception of specific operations by the infrastructure owner. To avoid tight coupling between component providers and the infrastructure owner, we limit the interception activity to SPOT API calls and event interaction of application components.

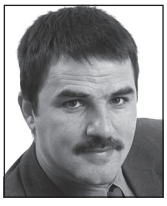
About the Authors

Christophe Huygens is a part-time guest professor in the Department of Computer Science at Katholieke Universiteit Leuven in Belgium and member of its Distrinet research group. He's also a private consultant for network and systems security and operational risk-control tools. His research focuses on policy and risk, and how to manage, monitor, enforce, and control policy from an infrastructure perspective. Huygens has an MSc in both computer science and electrical engineering from Katholieke Universiteit Leuven. Contact him at christophe.huygens@cs.kuleuven.be.

Danny Hughes is a lecturer in the Department of Computer Science and Software Engineering at Xi'an Jiaotong-Liverpool University and a free associate of the Distrinet research group. His research focuses on providing middleware for wireless sensor networks and their application to problems in the environmental-monitoring domain. Hughes has a PhD in computer science from Lancaster University. Contact him at daniel.hughes@xjtlu.edu.cn.

Bert Lagaisse is a postdoctoral researcher in the Distrinet research group at Katholieke Universiteit Leuven. His research interests include distributed systems, enterprise middleware, component frameworks, and aspect-oriented software development. Lagaisse has a PhD in computer science from Katholieke Universiteit Leuven. Contact him at bert.lagaisse@cs.kuleuven.be.

Wouter Joosen is a full professor in the Department of Computer Science of Katholieke Universiteit Leuven. His research interests include aspect-oriented software development (focusing on software architecture and middleware) and security aspects of software, including security in component frameworks and security architectures. Joosen has a PhD in computer science from Katholieke Universiteit Leuven. Contact him at wouter.joosen@cs.kuleuven.be.



Several challenges remain. Unrelated to the use of multiple paradigms, we've identified frequent changes in state and objectives due to the impact of networked embedded systems' dynamics and their impact on the planning process. Keeping the planning feedback loop stable while looking for optimum allocation based on out-of-date and/or incomplete state information is difficult. Another issue is the need for planning coordination among multiple infrastructure owners when applications are truly federated.

Specific to the use of multiple paradigms, more research is needed in several areas. First, the automatic choice between paradigms later in the development chain isn't always obvious. For example, whereas business goals result in runtime component artifacts, nonfunctional goals such as security can be realized by using specific security components but also through policy-driven mechanisms. Second, although researchers have benchmarked traditional networked-embedded-systems evolution by simulating improvements—for example, for subproblems such as efficient routing—it's difficult to quantitatively demonstrate the benefits of using multiple paradigms in the development process. Finally, the identified actors don't work in silos; they all influence the same continuum. So, infrastructure owner actions can cause side effects that violate assumptions made by component developers. Using different paradigms complicates this picture.

Regardless of the exact paradigms used, however, we've made substantial progress in improving the application and usefulness of networked embedded systems through a broader interpretation of development activities. Our interpretation of multiparadigm development has coevolved with this process. We're convinced that the presented perspective will be of value to more traditional distributed-computing environments as well.

Acknowledgments

This research was funded by the Interuniversity Attraction Poles Programme of Belgian State, Belgian Science Policy, the Interdisciplinary Institute for Broadband Technology, and the Katholieke Universiteit Leuven Research Fund.

References

1. P. Anderson, *System Configuration*, *Short Topics in System Administration*, vol. 14, Usenix Assoc., 2006.
2. D. Hughes et al., "LooCI: A Loosely-Coupled Component Infrastructure for Networked Embedded Systems," *Proc. 7th Int'l Conf. Mobile Computing and Multimedia Conf.* (MoMM 09), ACM Press, 2009, pp. 195–203.
3. N. Matthys et al., "Fine-Grained Tailoring of Component Behaviour for Embedded Systems," *Proc. 7th IFIP WG 10.2 Int'l Workshop Software Technologies for Embedded and Ubiquitous Systems* (SEUS 09), LNCS 5860, Springer, 2009, pp. 156–167.
4. T. Delaet and W. Joosen, "PoDIM: A Language for High-Level Configuration Management," *Proc. 21st Large Installation System Administration Conf.* (LISA 07), Usenix Assoc., 2007, pp. 261–273.