TITLE

Streamlining development for Networked Embedded Systems using multiple paradigms

AUTHORS

Corresponding Author:

Christophe Huygens
Department of Computer Science
Katholieke Universiteit Leuven
Christophe.Huygens@cs.kuleuven.be
Celestijnenlaan 200A
B-3001 Heverlee BELGIUM
fax: ++32 (0)16/32.79.96
tel: ++32 (0)16/32.77.00

Co-Authors:

Danny Hughes
Xi'an Jiaotong-Liverpool University (XJTLU)
daniel.hughes@xjtlu.edu.cn

Bert Lagaisse
Department of Computer Science
Katholieke Universiteit Leuven
bert.lagaisse@cs.kuleuven.be

Wouter Joosen
Department of Computer Science
Katholieke Universiteit Leuven
Wouter.Joosen@cs.kuleuven.be

VERSION - 2.0

DATE OF REVISION – 10 April 2010

ABSTRACT

The limitations of network embedded systems imposed by mobility and adaption scenarios in a context of scarce energy and system resources mandate optimization throughout the life cycle of the hardware and software combination. Deployment, operational activities and maintenance require software development activities that must all be aligned to become integral parts of the development process. Yet the diversity of the actors participating in the life cycle must be acknowledged. We have identified the following activities to specify behavior throughout the life cycle phases of networked embedded systems:

- Traditional software development according to component based engineering principles that deliver business and application logic.
- Development of pre-deployment artifacts using aspect-oriented composition.
- Development of deployment artifacts using declarative abstractions to describe the goals of the actors and the application compositions.
- Development of specific run-time artifacts using declarative or imperative policies.

The notion of multiparadigm programming in this paper addresses an integrated approach to the development of complementary artifacts that are all essential, but unfeasible to be programmed using one single paradigm. The article focuses on i.) the proposed actors contributing to the development phases, ii.) their activities in terms of methods and artifacts and iii.) a concise discussion on the findings.

KEYWORDS

D.2.17.i	Programming paradigms
D.2.17.j	System integration and implementation
J.9.f	Wireless sensor networks
K.6.2.c	Computing equipment management

ABOUT THE AUTHORS

Christophe Huygens

Christophe Huygens is Part-time Guest Professor at the Department of Computer Science of the Katholieke Universiteit Leuven in Belgium and member of its DistriNet research group. His focus is on policy and risk, and how to manage/monitor and enforce/control policy from an infrastructure perspective. He also consults privately on network and systems security and operational risk control tools.

Danny Hughes

Danny Hughes is a Lecturer at Xi'an Jiaotong-Liverpool University with the Computer Science and Software Engineering department and a Free Associate of the DistriNet research group. His research focuses on providing middleware for wireless sensor networks and the application of wireless sensor networks to problems from the environmental monitoring domain.

Bert Lagaisse

Bert Lagaisse is Postdoctoral Researcher at the DistriNet research group. He specializes in Aspect-Oriented composition in a middleware context. His earlier work centered around the comparison of J2EE and .NET, with focus on multi-tier architectures and security. Generally, he has a strong interest in distributed systems, enterprise middleware, component frameworks and AOSD. In his little spare time, Bert is an active .NET and Google Desktop Developer.

Wouter Joosen

Wouter Joosen is Full Professor at the Department of Computer Science of the Katholieke Universiteit Leuven in Belgium, where he teaches courses on software architecture and component-based software engineering, distributed systems and the engineering of secure service platforms. His research interests are in aspect-oriented software development, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.

Streamlining development for Networked Embedded Systems using multiple paradigms

The stringent requirements imposed by mobility and adaption scenarios in a context of scarce energy and system resources, typical for networked embedded systems, mandate optimization throughout the life cycle of the hardware and software combination. To enable this optimization the life cycle of an embedded system must be addressed in an integrated way, avoiding a single-application development focus. Deployment, operational activities and maintenance require software development activities that must all be considered to become integral parts of the software development continuum. One cannot afford the traditional rift between software development, system engineering and system management. Yet the actors of the continuum are highly diverse with respect to life cycle phase participation, goals, typical processes and skills and the integrated approach must respect this diversity.

From this perspective, the notion of multiparadigm programming in this article addresses an integrated approach to the development of complementary artifacts that are all essential in the full life cycle of a distributed embedded system, but infeasible to be programmed using one single paradigm. This challenging new setting raises several questions regarding the organization of software development. Who are the typical stakeholders in a solution using networked embedded systems? How can the development task be divided amongst these process actors? And, from a development perspective, what are appropriate programming approaches that align not only with the objectives, but also the skill set of the actor. We have identified, applied and validated component-based development, policy-driven administration and constraint-based goal declarations - at multiple abstraction levels - to be of great and complementary value in this context. In what follows, we present lessons-learned based on our research activities and prototypes developed in cooperation with industry partners.

Key activities and paradigms for networked embedded systems development

- *The (obvious) traditional software development that delivers business and application logic: we typically develop communicating software components according to **component based software engineering** principles.*
- *Development of pre-deployment artifacts: use of **aspect-oriented composition** results in instrumentation of the components or their communication by selective insertion of policy engines that allow cross-cutting concerns such as security or optimization to be managed more dynamically.*
- *Development of deployment artifacts: **high-level, declarative abstractions** describe the goals of the various actors as well as 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 run-time artifacts that can affect the application logic at execution time as the behavior of the running system can be externally influenced by **declarative or imperative policy** definitions.*

Stakeholders of the system-software continuum

Networked embedded systems of the future will be federated infrastructures where resources are shared amongst different applications. The **infrastructure owner** controls his or her pool of devices making up part of the federation. Applications may span multiple administrative boundaries, reflecting the business goals of the **application owner**. Prepackaged functionality to support the goals of the infrastructure and application owners will be provided by **component developers** and is made available in repositories. When functionality is unique to the application, the application owner must develop it.

Life cycle of a networked embedded system application

The application owner will compose the application in terms of existing or new components. The application owner will then submit the composition (and potentially new component code) to the infrastructure owner, together 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 some application owners. The composition and actors' goals are input to the **deployment phase**. The objective of this phase is to merge all interests, look for a solution that adheres to a maximum of goals, and translate the goals into actual scenarios that can be deployed. This scenario is then executed in the **run-time phase**. At runtime substitution of application elements is possible, and it is desirable for the infrastructure owner to have access to mechanisms that can influence the ensemble of running applications and thus fine-tune system functionality. The deployment phase can be preceded by **pre-deployment** activities in which the infrastructure owner can process and manipulate the components in any way he or she sees fit. Within a single networked embedded system, as viewed from a multi-application perspective, at any given time some applications may be running, while others are in the pre-deployment and deployment phases.

Building management case – stakeholders, life cycle, development activities and artifacts

*Many applications are running on the stationary nodes in a hospital. We can identify the following **application owners**: building management with physical access and HVAC applications, business support functions such as equipment asset tracking or even medical functions such as group patient monitoring. In addition to these existing applications, because of recent developments, the medical application owner wants to deploy a flu monitoring system. This system reuses the existing infrared capabilities of cameras connected to the building nodes, used for physical security and fire prevention, to assess patient temperature. To build this functionality, the medical application owner composes the application based on existing components in the system (as provided by **component developers**), such as image-based person detection and temperature estimation, but provides his or her own decision support system. This*

new application does not need to run in every room – a statistically significant sample is sufficient. The new components, the composition and the quality goals, such as desired coverage statistic are now submitted to the **infrastructure owner** – in this case the hospital ICT department. During the **pre-deployment phase** the ICT department will analyze the components from various perspectives. For example respecting security concerns will mandate communications security for patient data and critical components, such as image-based person detection, will be access-controlled. The quality objectives of the flu monitoring application (minimum coverage: 2 rooms) are then merged with objectives of the infrastructure owner (the fire-control system has absolute priority) and the flu application's composition to come to an actual **deployment** of instrumented components and associated wiring. At **run-time** the flu application can still be changed, controlled and augmented: better detection algorithms can become available resulting in substitution, or an outage of several nodes can cause this long term, non-critical monitoring function to be inactivated or even undeployed because of limited resources.

Artifacts as produced by A. Component developers B. Application Owner C. Infrastructure owner and D. The plan: components, wirings and policies ready for deployment (note weaving integration strategy) (Numbers refer to activities in article Figure 1, P = Policy artifact)

Development activities and programming approaches

The needs of the actors in the development process are diverse, as is currently reflected in the divergent approaches that system and software engineers use. To achieve good results, network embedded developers need to be system aware, whilst their infrastructural counterparts need to step up from low-level configuration management to an abstraction level adequate for cross-actor optimization [1]. Figure 1 shows the approach we have been advocating.

Development activity =
specification of behavior
in the Lifecycle Phase

↓

Lifecycle Phase		Actor	Component Developer	Application Owner	Infrastructure Owner
Pre-Deployment	Abstraction Level		N/A	N/A	High
	Development Activity				⑤ Declarative aspect-oriented specification of component composition
Deployment	Abstraction Level		N/A	High	High
	Development Activity			② Declarative constraint-based specification of application qualities ③ Declarative specification of the component composition	⑥ Declarative constraint-based specification of system-wide qualities
Run-time	Abstraction Level		Low	Low	Mid
	Development Activity		① Component-based development	④ Component-based development	⑦ Declarative or imperative rule-based specification of policies

Figure 1: Actors and development activities for networked embedded systems

Development activities targeted at the runtime phase

Given the limited capabilities of WSN platforms, and the emphasis on sense-process-and-send, most functionality will already be available. A **component-based programming model** offers the

embedded system developer reuse of existing artifacts, a flexible deployment and wiring model and does not dictate a specific host language.

The component-based approach is not attractive to infrastructure owners. Concerns exist that are hard for component developers to recognize or must be realized independently (for example because of frequent change). For example, in a building management scenario, we may temporarily want to suspend all components in order to devote resources exclusively to the fire detection and control component. For an example of the second class, we need to make sure the insulin pump of a patient is never actuated by an HVAC control component. To some extent these concerns could be addressed at the composition or deployment level, but acting there is expensive in the network embedded environment where resources are at a premium. Offering the infrastructure owner a dynamic mechanism to fine-tune run-time 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 sysadmin view on infrastructure [1].

Development activities targeted at the pre-deployment phase

The policy engine can be included in the middleware so that all interactions with other components must pass the policy engine. This strategy has two key disadvantages. The middleware is bloated with potentially unused functions and omnipresent policy evaluations entail overhead. A more elegant solution is to allow granular inclusion of the policy engine by the infrastructure owner during the pre-deployment 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 to describe concerns that are not policy-driven but that can be implemented by including other components and/or that influence the wiring between components in a manner similar to deployment specifications.

Development activities targeted at the deployment phase

Low-level reprogramming of networked embedded systems is impractical due to scale, heterogeneity and node mobility. Development in this phase equals describing the objectives of the stakeholders using high-level abstractions ultimately resulting in loading and wiring of components, policies and their associated code. The **quality objectives** of the application, such as desired coverage or timeliness of data are best described **declaratively using constraints**. The same applies to the quality objectives of the system, where concerns such as precedence between applications or overall system lifetime optimization come to mind. Next to the quality objectives of the application, its core business logic must also be described. In component programming, **composing** boils down to a **declarative configuration** of links between the provided and required interfaces of the components involved.

Different Middleware

Two observations can be made regarding our approach. First, it is clear that the traditional sequence of: development $\Rightarrow$ deployment $\Rightarrow$ operations is not strictly adhered to. Without any component development activity, the pre-deployment, deployment and run-time phases can be re-initiated because of simultaneous development activities or system dynamics. Therefore, we claim that it is appropriate to consider programming for the complete life cycle continuum rather than having isolated development silos. On the other hand, the multiparadigm approach in this case must not only be facilitated by platform, development tools and IDEs, but mainly by next generation middleware that provides the integrated execution environment for the various types of artifacts.

Programming with the DisSeNT middleware on the Sun SPOT Platform

In this section we present our DisSeNT (DistriNet Sensor Network Toolkit) middleware as an illustrative case of the presented approach to embedded development. Figure 2 revisits the various development activities in the context of our middleware solution based on the Sun SPOT platform [2]. Sun SPOT (Sun Small Programmable Object Technology) is a wireless sensor network device based on the Squawk Java Micro Edition Virtual Machine.

Lifecycle Phase		Actor	Component Developer	Application Owner	Infrastructure Owner
Pre-Deployment	Abstraction Level		N/A	N/A	Low
	Development Activity				Byte code instrumentation of Java ME target code using AspectJ
Deployment	Abstraction Level		N/A	High	High
	Development Activity			Domain-specific language for coverage qualities Declarative directed XML graphs for compositions	Object oriented domain modeling and constraint rules (declarative policy)
Run-time	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)

Figure 2: Development activities for DisSeNT middleware on Sun SPOT sensors

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 that provides standard component manipulation methods (such as **start**, **stop**, **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 **Jave ME** application into a component only requires base class modification and an **init()** call. Components publish and subscribe events. Component wiring translates to initiation of 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 performance overhead also validate feasibility on small platforms [3]. While the use of Sun SPOT automatically implies Java and an object-oriented paradigm, these are in no way mandatory for our approach. The key run-time paradigm is the component concept and the associated event based interaction.

To offer the infrastructure owner a suitable run-time abstraction, **Event-Condition-Action rules** are used. This aligns well with the event based communications model and the more imperative approach is well established amongst systems engineers. The planner can generate most ECA policies automatically from the deployment specifications (see below), but we also facilitate control by the system owner through provision of support tools to specify, check and deploy policies. Analysis of policy language, code size, resource consumption and overhead of the corresponding middleware can be found in [4]. The Action part of the rules is diverse: events can be manipulated, other events initiated, redirected to other components, or methods called. The listings underneath demonstrate how a typical non-functional requirement such as channel security can be integrated without involvement from the application owner or component developer. As **LOCATION** events are generated and consumed, the security primitives are applied and the event passed on to its original destination:

```
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;
    )}

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;
    )}
```

Deployment phase

Deployment related activities in our solution mostly use a **domain specific language with support to express constraints** that specify qualities. Expressing constraints inevitably involves the definition of metrics to specify intended behavior. In the context of network embedded systems, our target is either a node or a group of nodes. For a group we specify intended coverage. However, given the unreliable and dynamic nature of the target, a fully covered, atomic deployment is difficult to achieve – in contrast to more traditional systems, deployment takes time to propagate in networked embedded systems and full coverage may never be achieved. We further detail the deployment by “consistency metrics”, where we express consistency of deployment in terms of completeness and timeliness. Alternatively, we explored modeling the infrastructural domain using object oriented techniques [5], and subsequently applying constraint-based rules to the modeled universe in order to express goals.

Compositions are described by the application owner using a **digraph relating components** either from the catalog or to be provided by the application owner. The middleware now needs to enact the declarative deployment definitions, a process composed of several (semi-automatic) steps resulting in an actual deployable configuration or “plan”:

1. The first step is that some form of precedence as set by the infrastructure owner resolves conflicting objectives. For example, security and performance objectives may be conflicting.
2. To satisfy each objective a tactic must be chosen. For example, security can be implemented through the policy mechanism, while data accuracy may be obtained by duplication of measurement components.
3. A solution set must then be found that is translated to the deployment plan, a series of targeted `deploy_component()`, `wire_component()` or `deploy_policy()` instructions.
4. The above steps must be re-iterated at periodic intervals as applications are added, application or system objectives change or state changes happen in the nodes due to mobility or resource consumption.

Since this advanced deployment problem can be largely dealt with in the resource-rich backend tier of the WSN architecture, complex scheduling and planning algorithms may be applied with little performance impact for the networked embedded system.

Pre-deployment phase

Our middleware at this time does not handle aspect oriented composition declarations. The concerns still need to be detailed at a low abstraction level. However, simple template-based translation from declaration to implementation artifacts appears realistic for some concerns. To provide for the aspect-oriented instrumentation by the infrastructure owner **AspectJ** is used, merging the Java Micro Edition and AspectJ tool chains. Again, since instrumentation is done in the resource-rich backend tier, the performance and resource consumption of this activity is not

an issue. The AspectJ run-time that is required for the weaving operation is large but can be stripped down to a few kB before inclusion into the deployment unit by deleting all but the exception handlers. The code snippet below shows how the infrastructure owner can intercept specific operations in submitted code, albeit at the very low abstraction level.

```
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();}
    }
}
```

An obvious use is the redirection of resource access to the ECA-driven policy engine. To tackle the problem of tight coupling between the artifacts produced by the different actors we take care not to apply aspect oriented programming in the universal sense but limit its application to functionality well known to component, application and infrastructure actors. Examples are SPOT application-independent API calls or component communication over the event bus.

Challenges and conclusions

Several challenges remain. Not related to the use of multiple paradigms, we identified the presence of frequent state and objectives changes due to the dynamics of the networked embedded system and its impact on the planning process as problematic. 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 amongst multiple infrastructure owners when applications are truly federated.

Specific to the use of multiple paradigms more research is needed in the following areas:

- The (automatic) choice between paradigms later in the development chain is not always obvious. For example, while business goals result in run-time component artifacts, a non-functional concern such as security can be realized by using a specific security component but equally through a policy-driven mechanism.
- While traditional networked embedded systems evolution has been benchmarked by simulating improvements, for example for sub-problems such as efficient routing, it is hard to quantitatively demonstrate benefits of using multiple paradigms in the development process.
- The identified actors do not work in silos. They all influence the same continuum. So, infrastructure owner action results in side effects that violate assumptions made by component developers. The use of different paradigms complicates this picture.

Regardless of the exact paradigms used, we have made substantial progress in improving the application and usefulness of networked embedded systems by a broader interpretation of the development activity. The notion of multiparadigm development has co-evolved with this process. We are convinced that the presented perspective will be of value to more traditional distributed computing environments as well.

References

1. P. Anderson, "System Configuration". Sage, Short Topics in System Administration, #14, the USENIX Association, 2006, ISBN 1-931971-42-0.
2. Sun SPOT, <http://www.sunspotworld.com>.
3. D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, W. Joosen, "LooCI: A loosely-coupled component infrastructure for networked embedded systems", *Mobile computing & Multimedia*, Kuala Lumpur, Malaysia, 14-16 December 2009.
4. N. Matthys, D. Hughes, S. Michiels, C. Huygens, W. Joosen, "Fine-grained tailoring of component behaviour for embedded systems", *The 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, LNCS 5860, Newport Beach, CA, USA, November 16-18, 2009, pp. 156-167.
5. T. Delaet and W. Joosen, "PoDIM: a language for high-level configuration management" *Proceedings of the 21st Conference on Large Installation System Administration Conference* (Dallas, November 11 - 16, 2007). P. Anderson, Ed. USENIX Association, Berkeley, CA, pp 1-13.
6. M. Wang, J. Cao, J. Li, and S. Das, "Middleware for Wireless Sensor Networks: A Survey", *Journal of Computer Science and Technology*, 23(3), 2008, pp. 305-326.
7. J. Yick, B. Mukherjee, and D. Ghosal, "Wireless Sensor Network Survey", *Computer Networks*, 52, 2008, pp. 2292-2330.

Wireless Sensor Networks

Wireless Sensor Networks [6,7] are a class of network 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. WSNs are characterized by high mobility, minimal capabilities and low reliability. From a network perspective the nodes are mostly self-organizing, cooperating not only to achieve network-wide services such as routing or reliable node-to-node communication but also more complex services – for example data aggregation, localization or rogue node detection. These services are all highly optimized with respect to 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 making the links with the wired (Internet) infrastructure, where back-end infrastructure resides for management functions such as code deployment or resource allocation. Additional complexity is introduced in the deployment view by introduction of sharing (many applications want to use a limited amount of resources) or federation (where nodes having a limited trust relation cooperate towards an overall goal). Because of the highly dynamic nature of the environment, the need for high-level abstractions and the many concerns and actors involved, WSNs are ideally suited to test the limits of software engineering frameworks.