Supporting Reconfiguration and Re-use through Self-Describing Component Interfaces

Klaas Thoelen, Nelson Matthys, Wouter Horré,
Christophe Huygens and Wouter Joosen

Department of Computer Science,
Katholieke Universiteit Leuven,
3000, Leuven,
Belgium.

+32 16-327668

{firstname.lastname}@cs.kuleuven.be

Danny Hughes, Lei Fang and
Sheng-Uei Guan

Computer Science,
Xi'an Jiaotong-Liverpool University,
215123, Suzhou,
China.

+86 0512-8816-1508

{firstname.lastname}@xjtlu.edu.cn

ABSTRACT

Run-time reconfigurable component models have been highlighted as having particular potential in networked embedded systems. In these models, explicit interface definitions promote the re-use of generic units of functionality between application compositions, while run-time reconfiguration provides a mechanism to manage the dynamism of sensor network environments. Despite these advantages, in current systems, reliably re-using and reconfiguring distributed components is a complex undertaking. It requires a detailed understanding of the services offered by each component. The lack hereof effectively precludes run-time discovery and use of third-party components. This paper proposes the embedding of compact semantic descriptions in component interfaces and associated messages. These descriptions allow for efficient compatibility checking and therefore facilitate the run-time discovery and use of third-party component services. We demonstrate that this scheme is feasible in even the most resource-constrained sensor network environments.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Distributed Programming – component based software engineering.

General Terms

Algorithms, Design.

Keywords

Component based software engineering, networked embedded systems.

1. INTRODUCTION

A number of component models have been developed for networked embedded systems including NesC [1], OpenCOM [2], RUNES [3], and LooCI [4]. These component models have been used to implement distributed applications such as river

monitoring [5] flood prediction [6] and emergency response [3].

Embedded component models offer two key features that are particularly beneficial in networked embedded systems: run-time adaptation and software re-use:

- *Run-time adaptation* is used to match system behavior to changing environmental conditions in order to maintain optimal system operation. For example, GridStix use run-time reconfiguration to switch between networking approaches based upon environmental context and thus optimize network performance [6].
- *Software re-use* is promoted by concrete interfaces which promote the re-use of components between application compositions and minimize overhead due to component redeployment. For example, the Lorien Operating System allows for fine-grained system-level software updates, reducing overhead compared to monolithic mechanisms for evolving system functionality [7].

While the need for run-time adaptation and software re-use is particularly acute in resource constrained and dynamic environments, the complexity of re-using third-party software and managing distributed reconfiguration remains a key obstacle to exploiting these features.

This paper proposes that to address this, embedded component models should be extended such that all component interfaces provide a compact and computer-understandable semantic description of the service offered by each component interface. Such a description not only facilitates the manual discovery of third-party functionality, but also offers the possibility of autonomic reconfiguration. This paper describes and evaluates an efficient scheme for adding semantic information to component interfaces that is based on the work presented in [12]. We demonstrate that this scheme may be implemented with acceptable overhead even in resource constrained environments such as Wireless Sensor Networks (WSN).

The remainder of this paper is structured as follows: Section 2 provides an overview of related work in this area. We introduce our proposed system in section 3 and discuss the use of semantics throughout each stage of the component life-cycle in Section 4. Section 5 evaluates the scalability of our approach through simulation and Section 6 demonstrates how the ontology has been used in a real-world case study. Directions for future work are discussed in Section 7 before we conclude in Section 8.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MidSens'10, November 30, 2010 Bangalore, India.

Copyright 2010 ACM 978-1-4503-0454-2/10/11 \$10.00.

2. RELATED WORK

This section reviews three key areas of related work (i) embedded component models (ii) approaches to implementing distributed reconfiguration for networked embedded systems and (iii) semantic description languages. These are presented in sections 2.1 to 2.3 respectively.

2.1 Embedded Component Models

A number of run-time component models have been developed for networked embedded systems including OpenCOM [2], RUNES [3] and LooCI [4]. These component models allow developers to dynamically modify a running application without having to re-start the application. This is particularly advantageous in WSN scenarios, which are typically long-lived and need to adapt to changing environmental conditions. In addition reflection coupled with concrete interfaces supports the re-use of functionality across multiple application compositions.

The OpenCOM component model [2] is a generic, platform independent component model, which has been used to implement the GridStix WSN platform [8] and the Lorien Operating System [7]. In addition to components, OpenCOM offers the notion of ‘Component Frameworks’, which are discussed in more detail in section 2.2. Distributed bindings in OpenCOM follow RPC-like semantics.

The RUNES component model [3] is based on OpenCOM v2.0, compared to which it has a smaller footprint. RUNES also adds a number of introspection API calls to the OpenCOM kernel. The RUNES component model has been demonstrated in a tunnel fire application scenario [3]. RUNES is essentially a local component model offering no specific support for distributed component bindings.

The LooCI component model [4] was specifically designed for networked embedded environments and has been applied in a river monitoring application scenario [5]. In contrast to OpenCOM or RUNES, LooCI components follow an event-driven programming paradigm and LooCI offers support for distributed bindings over a common event bus communication abstraction.

All of the component models described above offer run-time support for introspection and reconfiguration. However, while the introspection services provided by these models may expose functional details, they do not provide a semantic description of the services offered by components, making it difficult to discover, reconfigure or re-use third-party components on the fly.

2.2 Distributed Reconfiguration

A key mechanism proposed by the research community for simplifying reconfiguration in distributed compositions is the notion of *component frameworks*.

OpenCom introduces the notion of Component Frameworks (CFs) [9], which are a special type of components that enforce local or distributed architectural patterns in application compositions. For example, the ‘Open Overlays’ CF [10] provides a generic pattern for implementing network protocols that consists of a cooperating set of three components: Control, State and Forward. As developers of diverse network protocols are encouraged to follow this pattern, the potential for re-use of these components between compositions is increased. As the CF is itself a component, it also simplifies reconfiguration, as calling methods on a CF will result in the operation being applied to all constituent components of that CF (for example, the developer could place all three overlay

components into quiescent state by calling the related operation on the Open Overlays CF).

While the use of component frameworks may facilitate distributed reconfiguration operations, this does nothing to address the re-use of deployed third-party functionality, as not all components that implement a role in the overlays CF are compatible. Our approach is thus highly complementary as we provide concrete semantic descriptions of component interfaces, which can be used to test for compatibility with a given composition.

2.3 Semantic Description Languages

Traditional semantic markup languages such as RDF [14] and OWL [15] have two critical shortcomings for describing resources available in networked embedded systems. The first key shortcoming is compactness. These schemes build upon XML and thus even simple semantic descriptions may be too large for highly embedded devices. In addition, while OWL-Lite descriptions may be efficiently parsed, simple operations such as subsumption and equivalence testing still incur needlessly high computational overhead.

The approach described in [12] by Preuveneers et al. uses the unique properties of prime numbers to generate more compact ontologies that support efficient equivalence and subsumption testing. We take a modified version of this approach as the foundation of our self-describing interface scheme.

3. SYSTEM OVERVIEW

In this section we introduce our approach and discuss its practical use. As defined in Section 1, the key obstacles to run-time adaptation and software re-use are:

1. **Discovery and re-use of third-party functionality.** The functionality or semantics of existing components developed by a third-party need to be known in order to make a well funded decision on its re-use in a new composition.
2. **Reconfiguration.** Correctly replacing an existing component with an updated version requires matching semantics between both components.

We tackle both problems by providing a compact and computer-understandable semantic description of the service offered by each component interface. To enable this, a coding scheme, based on the work in [12], is used to convert a hierarchical service taxonomy into a compact representation, which is used to tag component interfaces with the encoded service identifiers. The service taxonomy supports equivalence and subsumption testing in a resource-constrained environment. This enables inspection of 3rd party components by comparing the service identifiers of component interfaces and allowing reuse and reconfiguration when a match is found.

In the following sections we first elaborate on the service taxonomy and its encoding, followed by an architectural description and an overview of operation of the proposed system.

3.1 The Service Taxonomy

The service taxonomy is a tree-like data structure, the root of which is the base SERVICE type. In this data structure, child nodes are sub-types of their parent. For example, in the taxonomy shown in Figure 1, PRESSURE and ACCELERATION are both of service type SENSOR, however, PRESSURE is of service type

FLOOD, while ACCELERATION instead is of type THEFT (The meaning of the services is explained later on in Section 6).

An interface that offers, or consumes, a service type should accept all sub-types of that service. For example, a generic ‘pollution logging’ component with the receptacle POLLUTION may connect to components offering CONDUCTIVITY or METHANE interfaces, but *not* e.g. PRESSURE interfaces.

Figure 1: River Monitoring Service Taxonomy

The following data is stored in each node of the encoded service taxonomy:

- A unique human-readable **name** for this service.
- **seq**: A unique identifying number for this service that represents its position in the sequence of services as they were added to the taxonomy (i.e. the first service added will receive 1, the second service 2 and so on).
- **prime**: The n^{th} prime number where n is the sequence number **seq**. This is provided from a pre-computed list of the first 10,000 primes [17].
- **uid**: A unique identifier that encodes the position of this service in the service taxonomy in relation to its various ancestors. Multiplying the **uid** of the parent service with the **prime** of the incoming service generates this value. It is this value that allows for subsumption testing as described in section 3.3.1.
- **type**: A data type identifier that describes the encoding of data. Constants are provided for primitive types but URI links to WSDL descriptions may be used to describe complex types.
- **provider[]**: An enumeration containing URI links to all components in the repository that provide this service type.
- **consumer[]**: An enumeration containing URI links to all components in the repository that consume this service type.

3.2 System Architecture

The proposed architecture is shown in Figure 2 and makes a distinction between a back-end system and a sensor node.

In the back-end system, a *Network Manager* fulfills development and deployment tasks. It has access to the full *Service Taxonomy*, which is provided on a per-network basis. The *Component Repository* stores generic components and can be accessed by the Network Manager for deployment or by application developers who want to interact with existing components. The taxonomy scheme described in 3.1 imposes minimal requirements on the associated component repository. Specifically, it should expose all components with a specific URI. In a simple implementation, the repository could be a directory offered over the Internet on HTTP or FTP, or alternatively the Service Taxonomy may be used with more advanced software repositories such as UIMA [15] and PEEL [16].

A *Runtime Manager* on each sensor node controls the locally deployed components and allows the consultation of the semantic information of these components through a lookup table. To reduce the memory overhead at individual sensor nodes in the network, the full *Service Taxonomy* is not replicated on the nodes. In contrast, the *lookup table* only stores those properties of the taxonomy that are required by in-network operations. The *uid* is needed to support subsumption testing, while the smaller *sequence number* identifies the interfaces during other operations. Furthermore, only the interfaces of the components running on a specific node have an entry in its lookup table.

Figure 2: System Architecture Overview

3.3 Operation

The following sections 3.3.1 and 3.3.2 describe the operations that may be performed on the back-end taxonomy and on components at run-time respectively.

3.3.1 Interacting with the Back-End Taxonomy

The five key operations that may be performed on the taxonomy are presented in Table 1.

Table 1: Interface to the taxonomy

addServiceProducer (SEQ_NO, COMPONENT_URI)
addServiceConsumer (SEQ_NO, COMPONENT_URI)
addService (SEQ_NO, NAME, DATA_TYPE)
removeService (SEQ_NO)
subType (CHILD_UID, PARENT_UID)

The first two operations respectively add a URI link to the *provider* or *consumer* enumeration of an existing service for which *seq* equals to *SEQ_NO*. There is no return value.

The *addService*-method adds a service with the specified NAME and DATA_TYPE to the taxonomy as a child of the service with the matching *SEQ_NO*. It returns the *SEQ_NO* for the incoming event and the *UID* for this event. The incoming service must have the same DATA_TYPE as the parent service in order to ensure compatibility with existing components.

The *removeService*-method will remove a service from the taxonomy. In order to preserve the integrity of the taxonomy, services with children may not be removed.

The *subtype*-method will test if a *CHILD_UID* is a sub-type of *PARENT_UID*. This is achieved by dividing the value of *CHILD_UID* by the value of *PARENT_UID*. If the result of this operation has no remainder, then due to the unique properties of prime numbers, *CHILD_UID* must be a subtype of *PARENT_UID*. This mechanism is described in more detail in [12]. This method returns *true* if *CHILD_UID* is a subtype of *PARENT_UID*, otherwise it returns *false*.

3.3.2 Use of Semantic Data at Run-Time

The *lookup table* at each sensor node is used during: (i) component deployment, (ii) component discovery, (iii) component binding and (iv) binding inspection as described below.

Component Deployment: when a component is deployed, the component will register with the local entity that manages introspection, i.e. the Runtime Manager. The *uid* and *seq* of each interface will be entered into the local lookup table, allowing for the semantic description of this component to be discovered at run-time.

Component Discovery: when introspection functionality is called on a component, the Runtime Manager should return the *uid* value of each of its interfaces. The discovering entity may then perform subtype checking using the method outlined in Section 3.3.1 to ascertain whether the interfaces of that component are compatible with other elements of the composition.

Component Binding: when two interfaces are bound (i.e. connected) each interface should transmit its *uid* to the remote end-point. At this time, sub-type checking will be performed to ensure compatibility using the *uid* of the remote interface.

Binding Inspection: all messages that traverse component interfaces should contain the smaller sized *seq* of their originating interface. This allows data flows to be discovered by third parties. If necessary, these may also perform sub-type checking by introspecting the interface that produced the data in question.

It is important to note, that none of the interactions described above require access to the back-end ontology to discover and use compatible components. Instead, all of these activities may be supported in a distributed, decentralized fashion using the self-describing interface scheme.

4. ADDING AND USING SEMANTICS IN THE SOFTWARE LIFE-CYCLE

The following sections show how semantic descriptions of service interfaces play an important role in three stages of the software life-cycle: component development, application composition and

application deployment. We focus on the interaction with the *Service Taxonomy* and the *Component Repository* during each stage.

4.1 Component Development

In order to realize our vision of self-describing components, it is necessary for the developers of generic components to describe the services offered by each of their interfaces during component development.

To describe the services that the new component offers, the developer first browses through the taxonomy of existing interface services. If an existing semantic description matches the functionality of an interface that they provide, the developer will tag that interface with this service ID (*uid* and *seq*). The existing taxonomy entry will be amended to include the new component as a *provider*. Where a service does not exist, the developer will add a new child entry to the most logical parent node in the taxonomy and tag the component's interface accordingly. An abstract node may also be added to the taxonomy in order to describe a new group of services (see the 'POLLUTION' node in Figure 1 for an example of this).

When a developer creates a component that consumes a service, this service will be found in the taxonomy. The respective *uid* and *seq* will be used to tag the consuming interface and the existing taxonomy entry will be amended to include the new component as a *consumer*.

The developer will at this point have been provided with a *uid* and *seq* for each of the interfaces and should embed this data in each one of them (a component-model specific mechanism will be used). Once this process is complete, their component may be considered 'self describing'.

4.2 Application Composition

The role of the application composer is to develop a component connection graph (possibly supported by a graphical composition tool), which defines how components, producing and consuming various services, should be connected. The hierarchical semantic descriptions introduced in Section 3 play two important roles.

Firstly, when discovering components for use in the composition, the hierarchical service description provides an efficient mechanism for the developer to discover the services that they require, and the associated components that *provide* or *consume* these services.

Secondly, when connecting components, the subsumption checking service allows for automated verification of component compatibility, reducing the burden on the developer and helping to ensure the correctness of the resulting composition.

4.3 Application Deployment

At deployment time, semantic service descriptions are used to prevent the deployment of redundant functionality. This may be achieved manually, or more likely will occur automatically at the network gateway.

When presented with a composition for deployment, the deploying entity will use introspection to discover whether services of the type specified in the composition (or relevant sub-types) are already deployed at the required location. Where these components are already deployed and free to be reconfigured,

they may be used instead of deploying a new version of the component listed into the application composition. In this way, network overhead due to redundant component deployment can be automatically reduced or even eliminated.

5. SIMULATION

In order to demonstrate the efficiency of this scheme, we performed two sets of simulations to ascertain the worst-case size of a *uid* for taxonomies of different shapes and sizes.

In terms of *shape*, we generate test taxonomies where each node has a set number of children between 2 and 10. In terms of *size*, we first perform a small-scale simulation that shows how *uid* size increases for networks of up to 1000 services. We also perform a very large-scale simulation that shows how *uid* size increases for networks of up to 10,000 services. However, we believe that such large-scale networks will remain infeasible in the short term.

5.1 Small-Scale Simulation

Our ‘small-scale’ simulation features service taxonomies containing 200 to 1000 services (or unique interfaces). It should be noted that the smallest taxonomy we simulate contains more unique services than any distributed component composition described in the literature [5], [6] [10].

Figure 3: Worst-case uid size for a small-scale simulation

As can be seen from the figure, the size of the *uid* grows most quickly when nodes in the taxonomy have a small degree due to the increased number of multiplication operations performed in ‘skinny’ trees.

The worst-case *uid* size occurs where nodes have a branch factor of 2. In this case, the largest *uid* generated was 9 bytes. As this value only has to be transmitted at bind-time, and binding operations tend to be infrequent, we believe this overhead is acceptable.

5.2 Large-Scale Simulation

For the interest of the reader, we also provide a series of large-scale simulations that provide the worst-case *uid* size for taxonomies of 2000 to 10,000 services.

As can be seen from figure 3, the worst-case *uid* size for a network of 10,000 unique interfaces is 16 bytes. Again, as this value is transmitted only at bind time, we believe that the overhead incurred is acceptable, especially given the particular

advantages of semantically self-describing components in large networks. It is critical to note that an upper bound is placed on the scalability of this scheme by the availability of prime numbers. While the first 10,000 primes are readily available [17], too few are known to support much larger taxonomies. However, as previously stated, we do not believe that such a taxonomy space is required in the embedded systems.

Figure 4: Worst-case uid size for a large-scale simulation

6. A REAL-WORLD CASE-STUDY

To provide some real-world context for the simulations described in the previous section, we used the interface description scheme presented in this paper to describe a real-world LooCI [4] composition that was developed in a river monitoring application [5]. In this application, four sensor components provide input to an alarm-generating component. The conductivity and methane sensors monitor the pollution of the river, while the pressure sensor monitors the water level. Additionally, an accelerometer measures movements of the sensor nodes that might indicate an attempt of theft. An overview of the composition is provided in Figure 5.

Figure 5: River Monitoring Composition

The labels above each binding relate to corresponding taxonomy entry, which is shown in Figure 1. As before, every unique interface is assigned a unique semantic identifier.

As can be seen from Figure 1, in a small scale network of services, the resulting GUIDs are very small. The worst-case GUID is 266, which can be encoded in less than two bytes.

7. FUTURE WORK

The short-term focus of our future work will be on realizing an implementation of the scheme proposed in this paper for the LooCI component model [4].

Once this implementation is complete, we will evaluate the development overhead that adding semantic information imposes on the generic component creator along with the benefits that semantic descriptions provide to the application composer.

The degree to which this semantic information can be used to eliminate redundant component deployments is also a particular area of interest for us and will be explored using the LooCI implementation and real-world case-studies.

8. CONCLUSION

This paper argues that, in order to better support component discovery and re-use, component interfaces should be self-describing, providing a precise semantic description of the service offered by each interface. In order to support this vision, we have described a scheme for adding semantic information to component interfaces that is based upon the ontology description language introduced in [12].

Through a combination of large-scale simulation and a real-world case-study, we have demonstrated that the overhead entailed by using this scheme is reasonable.

9. ACKNOWLEDGMENTS

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

This research is conducted in the context of the IWT-STADIUM project No. 80037 [11]. Wouter Horré is a PhD fellow of the Research Foundation – Flanders (FWO).

The authors would like to thank Davy Preuveneers for his critical insights into how the scheme described in [12] could be used in this application domain.

10. REFERENCES

- [1] Gay D., Levis P., Von Behren R., Welsh M., Brewer E., Culler D., The NesC Language: A Holistic Approach to Networked Embedded Systems, in Proc. of the conference on Programming Language Design and Implementation, ACM SIGPLAN 2003, San Diego, California, USA, pp. 1 – 11.
- [2] Coulson G., Blair G., Grace P., Taiani F., Joolia A., Lee K., Ueyama J. and Sivaharan T., A Generic Component Model for Building Systems Software, in ACM Transactions on Computer Systems, Vol. 26, No. 1, Feb 2008.
- [3] Costa P., Coulson G., Gold R., Lad M., Mascolo C., Mottola L., Picco G.P., Sivaharan T., Weerasinghe N., Zachariadis S., The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario, in Proc. of the 5th Annual IEEE International Conference on Pervasive Computing (PerCom'07), White Plains, New York, March 2007, pp. 69 – 78.
- [4] Hughes D., Thoelen K., Horré W., Matthys N., Michiels S., Huygens C., Joosen W., Ueyama J., Building Wireless Sensor Network Applications with LooCI, in the International Journal of Mobile Computing and Multimedia Communications (IJMCMC), Vol. 2, No. 4.
- [5] Ueyama J., Hughes D., Matthys N., Horre W., Michiels S., Huygens C., Joosen W., An Event-based Component Model for Sensor Networks: a Case Study for River Monitoring, in the proc. of the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC'10).
- [6] Smith P., Hughes D., Beven K., Cross P., Tych W., Coulson G., Blair G., Towards the Provision of Site Specific Flood Warnings using Wireless Sensor Networks, in Wiley Inter-Science journal on Meteorological Applications, Vol. 16, No.1, January 2009, pp. 57 – 64.
- [7] Lorien: a pure dynamic component-based operating system for wireless sensor networks, Porter B., Coulson G., in proc. of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'09), Urbana Champaign, Illinois, USA, December 2009, pp. 7 – 12.
- [8] Hughes D., Greenwood P., Coulson G., Blair G., Pappenberger F., Smith P., Beven K., An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring, in the Inter-Science Journal on Concurrency and Computation: Practice and Experience, Vol. 20, No. 11, November 2007, pp. 1303-1316.
- [9] Grace P., Coulson G., Blair G., Porter B., Hughes D., Dynamic Reconfiguration in Sensor Middleware, in the proc. of the first International Workshop on Middleware for Sensor Networks (MidSens'06), Melbourne, Australia, November 2006, pp. 1-6.
- [10] Grace P., Hughes D. R., Porter B., Blair G. S., Coulson G., Taiani F., Experiences with Open Overlays: A Middleware Approach to Network Heterogeneity, in the proc. of the European Conference on Computer Systems (EuroSys '08), Glasgow, Scotland, UK, April 2008, pp. 123-136.
- [11] IWT Stadium project 80037, software technology for adaptable distributed middleware: <http://distrinet.cs.kuleuven.be/projects/stadium/>
- [12] Preuveneers D., Berbers Y., Encoding Semantic Awareness in Resource-Constrained Devices, in IEEE Intelligent Systems, vol. 23, no. 2, pages 26-33, March 2008, pp. 1541-1672.
- [13] OWL Web Ontology Language Overview, W3C Recommendation, available online at: <http://www.w3.org/TR/owl-features/>
- [14] RDF Semantic Web Standard, W3C standard, available online at: <http://www.w3.org/RDF/>
- [15] Götz T., Suhre O., Design and implementation of the UIMA Common Analysis System, in IBM Systems Journal, Vol. 43, No. 3, 2004, pp. 476 – 490.
- [16] Henninger S., Supporting the Construction and Evolution of Component Repositories, in proc. of the 18th International Conference on Software Engineering (ICSE'96), Berlin, Germany, 1996, pp. 279-288.
- [17] The University of Tennessee at Martin, The First 10,000 Primes, available online at: <http://primes.utm.edu/lists/small/10000.txt>