J Internet Serv Appl (2012) 3:277–290
DOI 10.1007/s13174-012-0064-0

SI: MIDDLEWARE '10 BEST WORKSHOP PAPERS

A reconfigurable component model with semantic type system for dynamic WSN applications

Klaas Thoelen · Danny Hughes · Nelson Matthys · Lei Fang · Simon Dobson · Yizhou Qiang · Wei Bai · Ka Lok Man · Sheng-Uei Guan · Davy Preuveneers · Sam Michiels · Christophe Huygens · Wouter Joosen

Received: 8 August 2011 / Accepted: 6 July 2012 / Published online: 23 August 2012

Abstract Runtime reconfigurable component models provide several attractions with regard to the management of wireless sensor network (WSN) applications operating in dynamic environments and under evolving application requirements. One such attraction is the runtime discovery of suitable components for reuse in changing application compositions. Syntactic interface typing, provided by contemporary component models, however only supports exact interface matching. This causes limited reuse of components and complicates management of WSN applications. We argue that more flexibility is required to efficiently manage the complex, large-scale and dynamic WSN deployments of the future. In this paper, we describe the addition of semantic service descriptions to component interfaces to support compatibility and subtype testing. This allows rich discovery and reuse of third-party functionality and reasoning at the level of equivalent service types. We report on the incorporation of these semantic interface definitions in the Loosely Coupled Component Infrastructure (LooCI). Evaluation thereof shows that the scheme imposes minimal computational and memory overhead, while significantly reducing the complexity and cost of reconfiguration.

K. Thoelen · D. Hughes (✉) · N. Matthys · D. Preuveneers · S. Michiels · C. Huygens · W. Joosen
IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
e-mail: danny.hughes@cs.kuleuven.be

K. Thoelen
e-mail: klaas.thoelen@cs.kuleuven.be

D. Hughes · Y. Qiang · W. Bai · K. L. Man · S.-U. Guan
Computer Science and Software Engineering,
Xian Jiaotong-Liverpool University, Suzhou 215123, China

L. Fang · S. Dobson
School of Computer Science,
University of St. Andrews, St. Andrews, KY16 9SX, UK

Keywords Component models · Semantic type system · Wireless sensor networks · Reconfiguration

1 Introduction

Component-based programming models [1–4] provide consistent encapsulation of software components, with explicitly defined functionality (i.e., provided interfaces) and explicit context dependencies (i.e., required interfaces). As components are well encapsulated and lack implicit dependencies, the application developer may safely compose third-party components into new applications, reducing the cost and complexity of application development. These characteristics promote software reuse and reduce development effort.

Those component models that offer support for runtime reconfiguration [2–4] allow application functionality to be modified at runtime through the insertion, removal, replacement and reconnection of components. This is particularly advantageous for wireless sensor network (WSN) applications that are subject to a high degree of dynamism and changing mission requirements. A number of academic papers have demonstrated the benefits of runtime reconfigurable component-based middleware in small-scale WSN scenarios with tens of nodes [5–7], and yet applying these models to build large-scale reconfigurable WSN applications remains complex.

Contemporary sensor networks may be composed of hundreds of nodes [8] and future applications such as Smart Cities are expected to scale to many thousands of nodes [9]. Each node in such scenarios may host several application components, each offering several interfaces. The scale, dynamism and complexity of such an environment, coupled with the possibility of network partition and intermittent connectivity render centralized control and management at the granularity of individual interfaces infeasible. This critical problem must be addressed before the benefits of reconfigurable component models can be fully exploited in large-scale WSNs.

This paper proposes a scheme to extend component models such that all component interfaces provide a compact semantic description of the functionality that they offer. This scheme facilitates manual reconfiguration by allowing the developer to reason at the level of functionally equivalent services, rather than unique interfaces, and supports autonomic reconfiguration through runtime compatibility testing of interface pairs. We have implemented this scheme for the Loosely Coupled Component Infrastructure (LooCI) component model [4]. Our evaluation shows that this scheme imposes minimal computational and memory overhead, while reducing the development complexity and bandwidth requirements of reconfiguration actions. In relation to our previous work [4,10,11], this paper offers the following unique contributions: (i) a complete description of the semantic type system of LooCI, (ii) a large-scale analytic evaluation and (iii) performance figures that quantify the overhead of semantics for two representative classes of WSN devices: AVR Raven [12] and Sun SPOT [15].

The remainder of this paper is structured as follows: Section 2 describes the LooCI, which provides our implementation environment. Section 3 describes the semantic type system and its role at development time and runtime. Section 4 provides a worst-case analysis of the algorithm used to encode semantic information. Section 5 provides a scenario-based evaluation on two classes of WSN hardware. Section 6 reviews related work; Sect. 7 discusses directions for future work. Finally, Sect. 8 concludes.

2 The LooCI middleware

We have implemented a prototype of our semantic type system for the Loosely-coupled Component Infrastructure (LooCI) [4]. LooCI is comprised of a lightweight execution environment, runtime reconfigurable component model and an event-based binding model. These elements are described in Sects. 2.1 to 2.3, respectively. Section 2.4 discusses reconfiguration in LooCI.

2.1 The LooCI execution environment

In this section, we introduce LooCIs execution environment. Figure 1 presents a distributed view of LooCIs architecture, providing details on the distributed event bus and a configured remote binding.

LooCIs *Reconfiguration Engine* maintains references to all local components and enacts incoming reconfiguration commands that are received over the event bus. As all

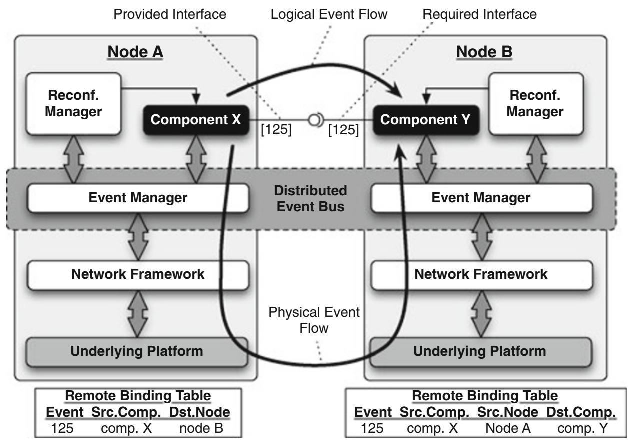

Fig. 1 The LooCI architecture

reconfiguration occurs over the event bus, it is possible for any component to enact reconfiguration of any other component within the network, subject to access control policies [17].

The *Event Manager* is a node local artifact that, cooperatively with instances on other nodes, implements a *Distributed Event Bus* to which every LooCI component and all Reconfiguration Engine modules are connected. Event bus communication follows a decentralized topic-based publish–subscribe model.

The *Network Framework* offers a uniform set of networking services to the upper middleware layers, including network-wide broadcast, one-hop broadcast and unicast. Components cannot access the Network Framework directly and communicate solely via explicit bindings over the event bus.

LooCI provides interoperability across various *Underlying Platforms*; i.e., operating systems and execution environments. Current implementations of LooCI allow developers to realize components using: C for Contiki [12], Java ME for Squawk [13] and Java SE for OSGi [15]. All ports of LooCI are open source [16].

2.2 The LooCI component model

The LooCI component model is platform and language agnostic, allowing developers to implement components in various languages and for different operating systems. Upon deployment, a LooCI component registers with the local Reconfiguration Engine, which supports introspection of component state and life cycle control.

LooCI offers a simple notion of component interfaces, wherein components model their provided interfaces as a set of events that may be published to the bus and their required interfaces as a set of events that may be consumed from the bus. In the LooCI component model, events always flow from provided interfaces to required interfaces. It is important to note, however, that the connection of required interfaces is optional. Components will neither send nor receive events until they have been bound and activated.

As all data is semantically typed and communication between components occurs exclusively via explicit bindings, introspection may be used to reify distributed relationships and reconfiguration may be used to modify data flows at runtime. The core LooCI API is provided in Listing 1 below.

Listing 1 The core LooCI API

```
// Deployment
CompID deploy(ComponentFile, NodeID)
Boolean removeComponent(CompID, NodeID)

// Control
Boolean activate(CompID, NodeID)
Boolean deactivate(CompID, NodeID)

// Introspection
CompID[] getComponents(NodeID, ComponentType)
String getComponentType(NodeID, CompID)
State getComponentState(NodeID, CompID)
Event[] getIfaces(NodeID, CompID, GUID)
Event[] getProvidedIfaces(NodeID, CompID, GUID)
Event[] getRequiredIfaces(NodeID, CompID, GUID)
NodeID[] getOutWires(NodeID, CompID, SHORT_ID)
NodeID[] getInWires(NodeID, CompID, SHORT_ID)

// Binding
Boolean wireFrom(Event, SourceCompID, SourceNodeID, DestCompID, DestNodeID)
Boolean wireTo(Event, SourceCompID, SourceNodeID, DestNodeID)
Boolean unwireFrom(Event, SourceCompID, SourceNodeID, DestCompID, DestNodeID)
Boolean unwireTo(Event, SourceCompID, SourceNodeID, DestNodeID)
```

2.3 The LooCI binding model

The LooCI event-bus is an asynchronous, event-based communication medium that promotes loosely coupled interactions. On the one hand, synchronization decoupling is provided by non-blocking interactions between components and the event bus. On the other hand, loose coupling in space is realized by separating distribution concerns from component implementation. LooCI components interact with the event bus via their provided and required interfaces, but have no knowledge about their communication partners. This information is stored in the binding tables of the Event Manager. A local binding table contains entries for bindings between local components, while a remote binding table contains entries for distributed bindings between a local and a remote component. The binding tables are therefore asymmetric. Bindings are only allowed between compatible interfaces and are generally defined by a *<source event type, source component ID, source address, destination event type, destination component ID, destination address>* tuple.

LooCI provides support for a rich set of binding modalities including: one-to-one, many-to-one, one-to-many, many-to-many and opportunistic (i.e., interactions that occur when nodes come within radio range). Section 2.4 describes the binding process.

2.4 Runtime reconfiguration in LooCI

Runtime reconfiguration and introspection in LooCI provides a mechanism to discover, integrate and remove software resources at runtime. As reconfiguration is enacted over the event bus, introspection and reconfiguration operations may be applied at various levels of granularity, targeting the entire network (*broadcast*), neighbors (*one-hop broadcast*) or specific nodes (*unicast*). All reconfiguration operations are implemented using the API provided in Listing 1.

- ***Software Service Discovery*** is achieved by sending a *getComponents* command to the target node(s), which respond(s) by returning the instance IDs of all deployed components or of all deployed components of a certain type if specified. The type of each component instance and its interfaces may be inspected using the *getComponentType* and *getIfaces* commands. This allows discovery of new software services at runtime.
- ***Integrating Components with Compositions*** is accomplished by issuing a *wireTo* command to the node hosting the provided interface and a *wireFrom* command to the node hosting the required interface. This establishes the necessary routing entries in the binding tables of the *Event Manager(s)* and thus connects the two components. Wiring commands may specify a specific component ID, or may use wildcards to connect compatible interfaces that are provided by unknown components.
- ***Removing Components from Compositions*** is accomplished by unwiring a component from a composition using the *unwireTo* and *unwireFrom* commands, which have identical syntax to their associated wiring commands. Where a component is no longer needed for any composition, it may be removed using the *removeComponent* method.

The LooCI core API contains the minimum functionality required to manage a network of LooCI nodes. While it may be used directly by an application composer, we expect it to be more commonly used by higher-level management tools that provide consistency and recovery services to distributed compositions.

The following sections describe the semantic type system and show how it is used to support the runtime reconfiguration activities outlined above.

3 The LooCI type system

The semantic type system proposed in this paper provides a single taxonomy that describes all software functionality that is available in a WSN. This formal, shared conceptualization of functionality allows for simplified discovery and type-safe

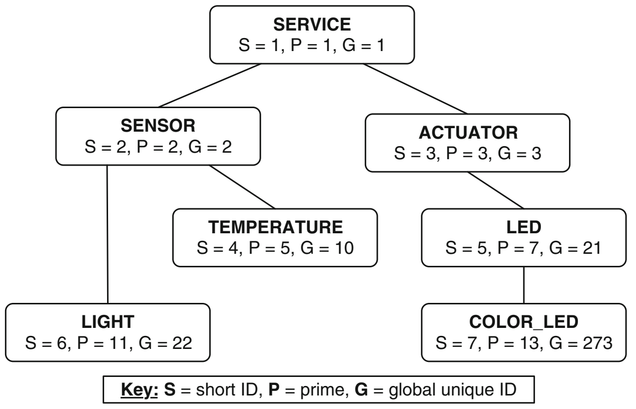

Fig. 2 Example software service taxonomy (Short ID, prime and global unique ID are discussed in Sect. 3.1)

reuse of third-party functionality between application compositions. As the LooCI type system describes component interfaces, rather than components, a single component may offer or require multiple software services from the taxonomy. The LooCI type system provides subtyping information and data formats for all software services.

The LooCI type system is encoded in a tree data structure, the root of which is the base SERVICE type. In the tree, all child nodes are subtypes of their parent. In the simple example taxonomy shown in Fig. 2, TEMPERATURE and LIGHT are of type SENSOR, however, service type LED is instead of type ACTUATOR. Any required interface that is compatible with a parent type is also compatible with all child types, and this allows the creation of generic components. For example, a generic *sensor-logger* component with the required interface SENSOR would be inherently compatible with provided interfaces offering the TEMPERATURE and LIGHT subtypes, but it would be incompatible with ACTUATOR or COLOR_LED as these are not subtypes of sensor.

Compatibility of parent and child types is ensured using a network-wide, daisy-chained message format. Each element in the taxonomy specifies 0 to n data items that must appear in messages associated with this software service and its children. Walking the tree from the root to a service and adding each specified data element sequentially to the message therefore derives the message format for a specific service. For example, if the ACTUATOR type specifies that its messages contain a boolean representing their activation state and the COLOR_LED type specifies that its messages contain a byte that represents the desired color, the complete message format of the COLOR_LED service is a boolean followed by a byte.

In the resource-rich back end, a full-fledged representation of the taxonomy is provided, while in the resource-impoverished sensor network, a highly optimized representation is used instead. Their design and implementation are described in Sect. 3.1 and 3.2, respectively.

3.1 The back-end type representation

The back-end representation of the type system is designed to facilitate component development. It allows *component developers* to easily describe new software services and link these semantic descriptions to concrete component interfaces in the component repository. *Application composers* may then use the service taxonomy to discover and use third-party components in their application composition.

The back-end representation of the taxonomy is encoded in an XML format to facilitate integration with Web Services and Enterprise Architectures. The following data is stored in each taxonomy node:

– ***NAME:*** A unique human-readable name that describes the type of service.
– ***SHORT_ID (S):*** A unique identifying number that represents the order in which the service was added to the taxonomy (i.e., the first service will receive *1*, the second service *2* and so on).
– ***PRIME (P):*** The n^{th} prime number where n is the SHORT_ID, used only to calculate the GUID.
– ***GUID (G):*** A unique identifier that encodes the position of this service in the taxonomy. Multiplying the GUID of the parent type with the PRIME of an incoming type generates the incoming types GUID.
– ***DATA_ITEMS[]:*** An array of DATA_ITEMs that must be included in messages of this type and all subtypes. Each item contains a simple data type (boolean, integer, byte, byte[] or string) and a text comment describing the contents of the item.
– ***PRO_IFACES[]:*** An enumeration containing URI links to all components in the repository with provided interfaces matching this type.
– ***REQ_IFACES[]:*** An enumeration containing URI links to all components in the repository with required interfaces matching this type.

The GUIDs encode the structure of the taxonomy using the unique properties of prime numbers. The quotient of a prime number has remainder zero only when divided by 1 or the prime number itself. It therefore follows that the quotient of the product of any set of primes has remainder zero only when divided by 1 or one of the primes contained within the set (we refer the reader to [10] for a formal proof of this property). In order to test whether GUID x is compatible with GUID y, where x is larger than y, we perform modulo (x, y). If the result is zero, then x must be a compatible subtype of y. If the result is non-zero, x is not a subtype of y. If GUID x is smaller than y, there is no need to evaluate for compatibility as it cannot be a subtype.

Consider the following example from the simple taxonomy provided in Fig. 2. The result of performing a modulo operation on any GUID in the taxonomy and SERVICE.GUID (1) is zero. Therefore, all service types are a subtype of SERVICE. The modulo of TEMPERATURE.GUID (10) and SENSOR.GUID (2) is also zero and thus TEMPERATURE is of type SENSOR. However, the modulo of LED.GUID (21) and SENSOR.GUID (2) is non-zero (1) and therefore, the LED service is not of type SENSOR.

Listing 2 API for the back-end type system

```
// Adding and removing services
SHORT_ID addService(SHORT_ID, NAME, DATA_ITEMS[])
Boolean removeService(SHORT_ID)

// Linking to components
Boolean addProvidedIface(SHORT_ID, URI)
Boolean addRequiredIface(SHORT_ID, URI)
Boolean delProvidedIface(SHORT_ID, URI)
Boolean delRequiredIface(SHORT_ID, URI)

// Discovering components
URI[] getComponentsProviding(SHORT_ID)
URI[] getComponentsRequiring(SHORT_ID)

// Querying the taxonomy
taxonomy getSubtree(SHORT_ID)
node getElement(SHORT_ID)
GUID getGUID(SHORT_ID)
Boolean compatible(CHILD_GUID, PARENT_GUID)
```

The back-end taxonomy representation imposes minimal requirements on an associated component repository. Specifically, such a repository must expose each component on a unique and persistent URI. In a simple implementation, the repository could be realized using an HTTP or FTP server. Alternatively, the taxonomy may be used with more advanced repositories such as UIMA [18] and PEEL [19]. An API is provided to maintain the type system, link it with the components in the repository and use it, as shown in Listing 2.

The LooCI type system plays an important role in component development and application composition. This is discussed in Sects. 3.1.1 and 3.1.2, respectively.

3.1.1 Type system in component development

It is anticipated that developers will first use the back-end taxonomy to locate components that offer similar services to those that they intend to develop. This prevents the developer from wasting effort in re-implementing functionally equivalent components, and provides context on the common model of service types that are currently used in the network. If existing components that can be reused are not discovered, the developer writes his component in the standard way, declaring provided and required interfaces as necessary.

At compile time, automated tools inspect the component and each interface is classified within the taxonomy. Where the service type is already described in the taxonomy, the existing GUID and SHORT_ID are embedded into the interface, and a reference to the interface is added to the taxonomy using either the *addProvidedIface* or *addRequiredIface* method as appropriate.

Where the software service represents a new type, the developer selects the most suitable parent element in the type system and adds the new service type to the taxonomy as a child of this element. A new GUID and SHORT_ID are created as described previously and embedded into the component interface. If required, the developer may also insert a new abstract type that will serve as a parent for their interface type. New services are added using the *addService* method, which specifies the SHORT_ID of the parent data type, a human readable NAME for the new type and DATA_ITEMS[] that should be included in associated messages. This may be extracted from the associated interface declaration. A reference to the associated interface is then added using either the *addProvidedIface* or *addRequiredIface* method as appropriate.

3.1.2 Type system in building applications

An application composer can use the back-end type system representation to find components to use in his composition. By navigating the type system taxonomy, he looks for the type of service that is required. Once found, he acquires a set of URI references to components with matching provided or required interfaces using *getComponentsProviding* and *getComponentsRequiring*, respectively. Using these URIs, components can be downloaded and composed into the application.

3.1.3 Querying and evolution of the type system

Four methods are available to query the type system at the back-end. *getSubtree* allows the user to retrieve the subtree underneath the type identified by SHORT_ID. *getElement* returns the taxonomy entry for the type denoted by SHORT_ID and *getGUID* returns the GUID associated with a given SHORT_ID. Finally, the *compatible* method provides a mechanism to test whether two types are compatible based on the specified parent and child GUIDs.

Evolution of the type system can occur during three phases: initial configuration, component development and maintenance. Firstly, during the *initial configuration* phase, the core types are defined as required by the LooCI Middleware and intended application domain. Secondly, during the *component development* phase, the type system expands by embedding component references into existing types and adding newly defined types as required by the developer (see Sect. 3.1.1). Thirdly, four *maintenance* operations may be performed on the type system: (i) *adding* new types, (ii) *pruning* leaf types without links to components, (iii) *restructuring* the type system, wherein the conceptual relations between types are altered and (iv) *optimization*, wherein the conceptual structure of the type system remains the same, but primes are re-assigned and GUIDs regenerated in order to minimize average-case GUID and therefore total taxonomy size. While *adding* and *pruning* of types may be performed without updating the in-network type representation, *restructuring* and *optimization* require recompilation and re-installation of the middleware on all participating nodes.

3.2 The in-network type representation

The in-network representation of the type system is designed to facilitate component discovery and the checking of type safety at runtime, while minimizing overhead. As the developers of the WSN interact with it exclusively through the back-end, there is no requirement to support in-network querying or maintenance of the type system. The in-network type representation therefore contains only the minimum information required to support compatibility testing.

As all component interactions occur via explicit bindings, compatibility checking need only be performed when discovering and binding to a component. As a result, each node only stores taxonomic data for the interfaces of locally deployed components, rather than the complete service taxonomy. This data consists only of the SHORT_ID and GUID for each interface that is hosted on a node. The remaining data from the back-end representation can be omitted for the following reasons. As the in-network taxonomy is optimized for automatic use, there is no need to include a human-readable service NAME. As GUIDs have already been calculated in the resource rich back-end, the PRIME can be omitted. A required interface that is compatible with a parent type is implicitly compatible with all child types due to the simple daisy-chained message format, so there is no need to include the DATA_ITEMS[] used in service messages. Finally, as software deployment only occurs from the back-end, it is unnecessary to include data that links types to components in the repository (i.e., PRO_IFACES[] and REQ_IFACES[]).

3.2.1 Distribution of type data in the network

Type data is injected into the network as required by components at deploy time. As described in Sect. 3.1.1, when the component developer declares an interface, the SHORT_ID and GUID of the associated type are embedded into the interface description. By consequence, each component becomes self-describing in terms of the services it requires and provides. Compatible services are discoverable using the *getIfaces* introspection commands, which will return references to any interface that is a subtype of the GUID specified.

As the GUID encodes the structure of the type system, it is significantly larger than the SHORT_ID, which is optimally compact. Therefore only the SHORT_ID is used when routing messages between interfaces: i.e., it is embedded in all messages and used in the binding tables of the Event Manager. On the other hand, GUIDs are only transmitted (i) with the component at deployment time, (ii) with introspection commands and responses to support discovery and (iii) with wiring commands to support type checking at bind-time. The overhead of the semantic type system at runtime can therefore be viewed as the overhead of transmitting additional type data and the computational overhead of compatibility testing.

3.2.2 Using the in-network type system at run-time

The in-network taxonomy is used at runtime to: (i) automatically check type safety on bindings, (ii) to discover compatible services and (iii) to reduce the overhead of configuration.

Compatibility testing of bindings reduces the scope for errors arising due to the binding of incompatible interfaces, in turn reducing overhead for the application composer. Compatibility testing is performed by the LooCI runtime whenever an interface is bound. If the types of the required and provided interfaces are incompatible, the binding does not occur and an error message is generated.

When the application composer or a software component discovers a software service using introspection (as described in Sect. 2.4), they may call the compatibility test method of the LooCI runtime middleware to determine if the service is compatible with their composition. This allows for the type safe discovery and use of compatible services, even where the specific components involved were unknown to the developer at build-time.

It is important to note, that none of the runtime interactions described above require access to the back-end taxonomy, or necessitate a human in the loop. However, the type system also supports the application composer in two important ways. Firstly, by discovering compatible deployed services, deployment of redundant components can be prevented. Secondly, in-network compatibility testing may be used to reduce configuration effort and message passing overhead during the application composition phase by allowing the application composer to refer to groups of interfaces by supertype.

4 Worst-case analytic evaluation

This section provides a worst-case analytic evaluation of the scalability of our taxonomy scheme.

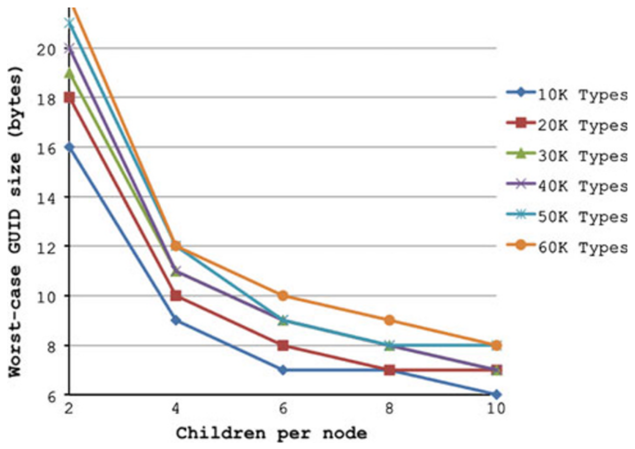

Fig. 3 Analysis of worst-case GUID size

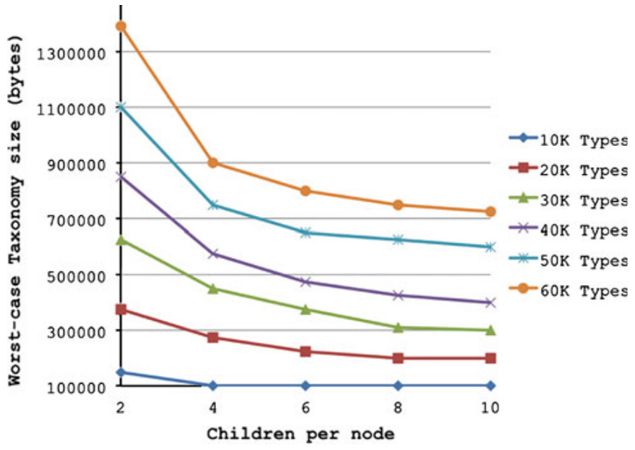

Fig. 4 Analysis of worst-case taxonomy size

In LooCI, SHORT_IDs are stored using two-byte identifiers, allowing up to 65,536 unique software services. We believe that this is ample as it allows for orders of magnitude more unique software services than reported in any of the current literature [4,6,7].

The number of bytes required to store the GUID is of great importance because it introduces overhead in terms of memory and communication (see Sect. 3.2.1). To demonstrate the efficiency of this scheme, we generated service taxonomies to ascertain the worst-case size of a GUID for taxonomies of different shapes and sizes. In terms of shape, we generated taxonomies where each node had a set number of children (i.e., degree) between 2 and 10. In terms of size, we generated taxonomies of 10,000 to 60,000 unique elements. As can be seen in Fig. 3, the size of the GUID is larger when nodes in the taxonomy have a small degree due to the increased number of multiplications that are performed in *skinny* trees. The worst-case taxonomy, with 60,000 elements and a branch factor of 2, has the largest GUID of 22 bytes. For a small increase in degree to 4, the worst-case GUID size drops to just 12 bytes.

We also evaluated the worst-case size of the taxonomy data that may be stored by a single node. This situation would arise when a single node is required to host interfaces of every type defined within the taxonomy. As can be seen from Fig. 4, taxonomy size is largest for trees of low degree, due to the larger GUIDs. In the worst case, the maximum taxonomy size generated was 1.36 MB. In our experiences and based upon the available literature [4,6,7], it is more likely that nodes will only host a small subset of service types, resulting in a much lower overhead as observed in the case study evaluation we present in Sect. 5.

5 A case study evaluation

We have evaluated the proposed type system in the context of a small-scale sensor network that is deployed in the roof garden of Xi'an Jiaotong-Liverpool University (XJTLU). The deployment monitors weather conditions and soil moisture and relays the gathered data to a back-end database and the web pad of a mobile user. Depending on available solar power, sensor data are either forwarded in near real time or batch fashion.

The following sections illustrate the role of our semantic type system throughout the life cycle of this application. Section 5.1 describes the event taxonomy that was created to support this application, which is discussed in greater detail in Sect. 5.2. Section 5.3 provides benchmark data on the performance of compatibility testing, while Sect. 5.4 measures the memory overhead that the type system induces on components. Section 5.5 analyses the overhead of the type system during runtime discovery. Finally, Sect. 5.6 discusses the benefits of the type system during configuration.

5.1 Scenario type taxonomy

The complete type taxonomy for this scenario is shown in Fig. 5. The root type EVENT is shown in black. Abstract types (with no implementing provided interfaces) are shown in white and concrete types (with at least one implementing provided interface) are shown in gray. This taxonomy contains a total of 67 elements, including 20 abstract types and 47 concrete types. The total size of the in-network taxonomy, encoded using the scheme described in Sect. 2, is 418 bytes. The sizes of the GUIDs range from 1 to 3 bytes.

It should be noted that the concrete types used in this taxonomy relate to a specific software command, response, physical sensor or actuator. This precise description of service types enables accurate reasoning using the type system. For example, while the BATT type may specify that all child types should provide the percentage of available battery energy, a distinction between SPOT_BATT and RVN_BATT is necessary to precisely understand the implications of a

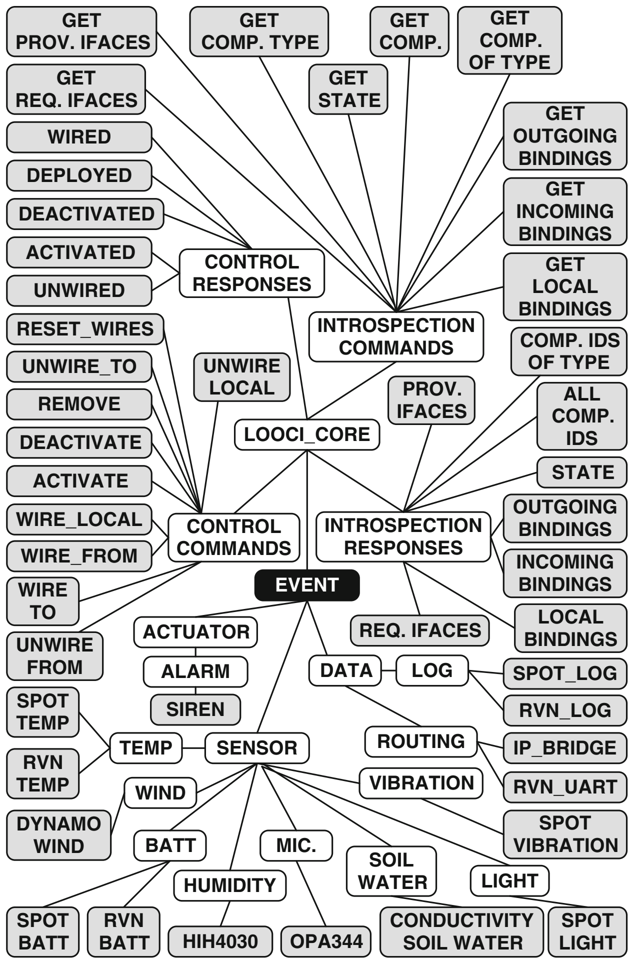

Fig. 5 Complete scenario taxonomy

given battery level due to the different battery capacities and power consumption characteristics of these motes.

5.2 XJTLU roof garden deployment

The system is composed of three types of sensor nodes and a base station. Four *AVR Raven* [12] sensor nodes are distributed throughout the garden and monitor soil moisture levels. The Raven nodes have a 16 MHz ATMega1284p CPU, 16 kB RAM, 128 kB flash memory and IEEE 802.15.4 networking. They run Contiki 2.4 [13] and LooCI [4]. Two *Sun SPOT* sensor nodes monitor weather conditions including temperature, light, humidity and wind speed. The SPOTs have a 180 MHz ARM9 CPU, 512 kB RAM, 4 MB flash memory and IEEE 802.15.4 networking. They run the SQUAWK JVM [14] and LooCI [4]. A single *Web Pad* provides a mobile interface to the system, allowing users to view sensor readings. The Web Pad has a 400 MHz XScale CPU, 64 MB RAM, 16 MB flash memory and IEEE 802.11b networking. They run embedded Linux, OSGi [16] and LooCI [4]. Finally, a single *Laptop*

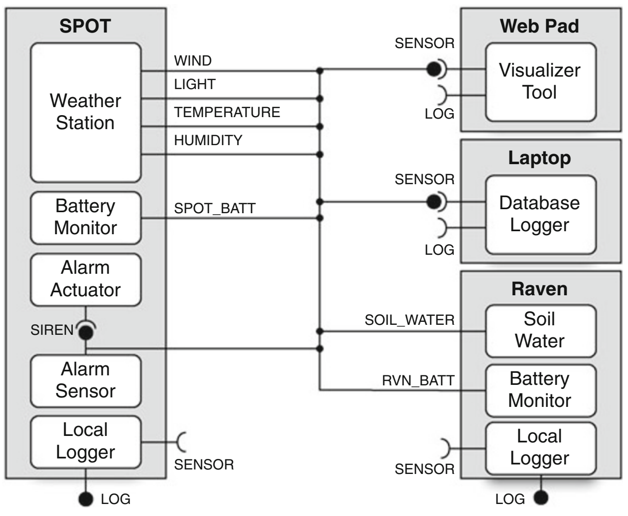

Fig. 6 Conceptual plentiful power composition diagram

serves as a base station and network bridge. It has a Pentium T400 CPU running at 2 GHz, 3 GB RAM, large hard drive, IEEE 802.11b and IEEE 802.15.4 networking. The Laptop runs Ubuntu Linux, OSGi [16] and LooCI [4]. While this scenario is small in scale, it demonstrates sufficient richness in terms of service types to evaluate how the proposed type system copes with a real-world scenario. The following LooCI components are used in this scenario (see Figs. 6, 7).

SPOT Software Components

Weather Station offers four provided interfaces for WIND, LIGHT, TEMPERATURE and HUMIDITY. All sensor readings are published at a rate of one per second.

Battery Monitor offers a single provided interface of type SPOT_BATT, which publishes data on available battery and solar power at a rate of one reading per second.

Alarm Sensor uses a vibration sensor and microphone to check for tampering. This component offers a single provided interface of type SIREN. When tampering is detected, a SIREN event is published.

Alarm Actuator offers a single required interface of type SIREN. Receipt of a SIREN event causes the siren to sound.

Local Logger offers a single required interface of type SENSOR. When an event is received on this interface, it is logged to flash memory. The component also offers a single provided interface of type LOG. Logged events are published in batch fashion on the LOG interface once every 30 s.

Raven Software Components

Soil Water offers a single provided interface of type SOIL_WATER. A conductivity probe provides soil moisture readings, which are published at a rate of one reading per second.

Battery Monitor offers a single provided interface of type RVN_BATT, which publishes data on available battery and solar power at a rate of one reading per second.

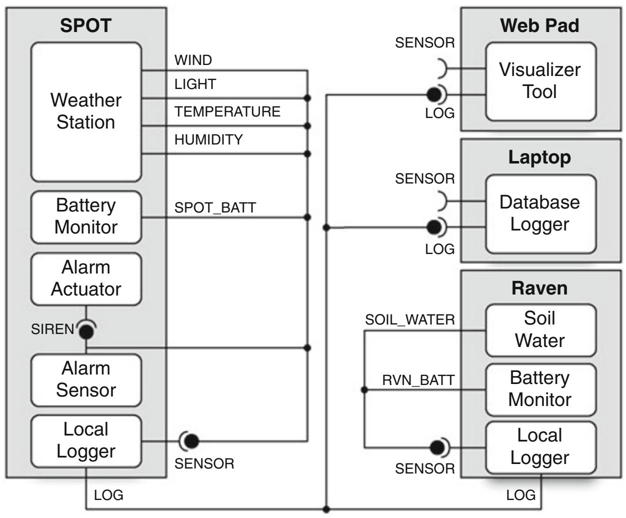

Fig. 7 Conceptual low power composition diagram

Local Logger offers a single required interface of type SENSOR. When an event is received on this interface, it is logged to flash memory. The component also offers a single provided interface of type LOG. Logged events are published in batch fashion on the LOG interface once every 30 s.

Web Pad Software Component

Visualizer Tool offers a required interface of type SENSOR, which receives single sensor readings, and a required interface of type LOG, which receives batches of sensor readings. All received sensor readings are displayed for the user in a simple GUI.

Laptop Software Component

Database Logger offers a required interface of type SENSOR, which logs single sensor reading events and a required interface of type LOG, which is used to log batch results. All received events are logged to a MySQL database for storage and later use.

As described in Sect. 2, LooCI provides a clean separation between a component and its bindings. The generic components outlined above, can thus be wired together by the application composer into various distributed application compositions. Furthermore, these compositions can be modified at runtime to suit changing application requirements or environmental conditions. In the context of this scenario, we use reconfiguration to adapt to changing power availability. When power is plentiful, the system is configured to relay environmental data to the base station in near real time, at a rate of one reading per second for all sensors. This configuration provides a high degree of timeliness at the cost of increased radio use and therefore reduced battery life. At the base station, environmental data are entered into a database for storage. This component configuration is shown in Fig. 6. When power becomes scarce, during periods with low solar power, the system reconfigures, such that sensor nodes log environmental data locally and relay data in batch fashion to the base station only once every 30 s. This configuration increases battery life by minimizing transmissions, though at the expense of timeliness. The component configuration is shown in Fig. 7.

Dynamic reconfiguration between these compositions at runtime is used to maximize performance when power is plentiful and maximize battery lifetime when power is scarce.

Table 1 Compatibility testing performance

	Avg. Compat. test time (ms)	Avg. local bind time (ms)	Overhead (%)
SPOT	0.29	24	1.21
Web Pad	0.05	3.6	1.35
Raven	0.03	0.30	9.93
Laptop	0.0004	0.24	0.18

5.3 Performance of compatibility testing

In order to assess the performance of compatibility testing on each of the devices in our scenario, we have performed exhaustive tests, wherein each element of the taxonomy described in Sect. 5.1 is tested for compatibility with all other elements. Each test comprised of 10,000 sequences of comparisons and was executed ten times. Table 1 shows the average-case time required for compatibility testing of taxonomy elements for each platform, along with the average time required to bind a LooCI component interface. The performance overhead of type safety at bind time is thus the additional time that is consumed in compatibility testing.

As shown in Table 1, for all devices used in this scenario, the average-case time required for compatibility testing is a fraction of a millisecond and compatibility checking adds between 0.18 and 9.93 % overhead on binding operations. We believe that this limited overhead is a small price to pay for type safety at bind time. The higher overhead of compatibility testing on the Raven may be attributed to the lack of hardware support for division on its Atmel 1284p microcontroller. We have furthermore evaluated the performance of multi-byte division on SPOT and Raven for growing GUIDs up to 8 bytes. This resulted in an acceptable worst-case processing time of 0.88ms on the Raven.

5.4 Memory footprint of the type system

This section provides context on the memory overhead introduced by the semantic type system on LooCI components. As mentioned in Sect. 3.2.1, each component embeds the SHORT_ID and GUID of its interfaces. The memory overhead of the hierarchical type system can thus be seen as the bytes required to store the semantic information contained in

Table 2 Memory requirements on SPOT

	Interfaces	Tax. data (bytes)	Comp. size (bytes)	Overhead (%)
Weather Station	4	12	2,963	0.40
Battery Monitor	1	3	2,610	0.11
Alarm Actuator	1	3	2,060	0.15
Alarm Sensor	1	3	3,062	0.10
Local Logger	2	3	169,091	<0.01
Total	9	24	179,786	0.01

Table 3 Memory requirements on Raven

	Interfaces	Tax. data (bytes)	Comp. size (bytes)	Overhead (%)
Soil Water	1	3	1,644	0.18
Battery Monitor	1	3	1,676	0.18
Local Logger	2	3	3,324	0.09
Total	4	9	6,646	0.14

the GUID. It should be noted that GUID size varies depending on position in the taxonomy as shown in Fig. 2. Those types that are higher in the tree tend to have a smaller GUID, while the leaf GUIDs tend to be larger, thus the per-interface overhead of taxonomic data varies.

As can be seen from Table 2, the worst-case memory consumption of the type system for any SPOT component in our application composition is 12 bytes for the weather station component, about 0.40 % of the total component size. In total, each SPOT stores 24 bytes of type data, which introduces a memory overhead of 0.01 % of the base component memory requirements.

As can be seen from Table 3, the memory consumption of the type system for any Raven component is 8 bytes, about 0.18 or 0.09 % of their respective component size. In total, each Raven stores 9 bytes of taxonomy data, which introduces a memory overhead of 0.14 % of base component requirements.

5.5 Overhead of the type system in discovery

LooCI provides support for the runtime discovery of components as described in Sect. 2.4 using the API specified in Listing 1. The semantic LooCI type system introduces messaging overhead, as GUIDs must be included in introspection messages to support discovery of compatible types. The *getIfaces* method of the LooCI API is supported by a response message containing a two-byte SHORT_ID, one byte component ID, the IPv6 address of the originating node and an array containing the SHORT_ID and GUID of all interfaces of the specified component that match the incoming query event. The overhead introduced by our semantic type system is hereby represented by the set of GUIDs. All other elements would also be required in case a non-semantic type system

Table 4 Service discovery overhead on SPOT (in bytes)

	Prov. Iface GUID	Req. Iface GUID	Other data	Taxonomy overhead (%)
Weather Station	12	0	11	109
Battery Monitor	3	0	5	60
Alarm Actuator	0	3	5	60
Alarm Sensor	3	0	5	60
Local Logger	2	1	7	43

Table 5 Service discovery overhead on Raven (in bytes)

	Prov. Iface GUID	Req. Iface GUID	Other data	Taxonomy overhead (%)
Soil Water	3	0	5	60
Battery Monitor	3	0	5	60
Local Logger	2	1	7	43

is used. Tables 4 and 5 show the overhead of the reply to a *getIfaces* query that matches against the root EVENT type (i.e., the worst case) for each component on the SPOT and Raven.

As can be seen from Tables 4 and 5, the semantic scheme used in LooCI results in a significant size increase for interface discovery responses, in our scenario up to 109 %. However, we believe that this is compensated for by (i) the ability to discover compatible subtypes and (ii) the increased efficiency of reconfiguration, as we demonstrate in Sect. 5.6. Furthermore, the increase of absolute number of bytes required is acceptable with regard to typical message sizes in WSNs (e.g., 127 bytes in IEEE 802.15.4).

5.6 Benefits of the type system in configuration

A key benefit of including semantic information in component interfaces is the ability to flexibly reconfigure services in the network, while conserving network overhead and reconfiguration effort. Consider the application composition described in Sect. 5.2. Where subtyping information is not present, separate wire commands must be issued for each interface that needs to be connected. However, by using supertypes the number of required commands can be reduced.

To illustrate this, consider the application composition shown in Fig. 6; it can be seen that all provided interfaces of subtype of SENSOR on the sensor nodes are wired to the Visualizer Tool and Database Logger on the Web Pad and Laptop, respectively. By using supertypes in the wiring commands, instead of separate subtypes, this reconfiguration can be enacted using far less wiring commands and therefore wire events.

Tables 6 and 7 summarize the number of events that must be sent to compose the plentiful power and scarce power

Table 6 Configuration overhead for plentiful power composition without semantics

	wire_to events	wire_from events	Total events	Total bytes
SPOT	13	1	14	295
Raven	4	0	4	84
Laptop	0	20	20	440
Web Pad	0	20	20	440
All Nodes	42	42	84	1,806

Table 7 Configuration overhead for low power composition without semantics

	wire_to events	wire_from events	Total events	Total bytes
SPOT	9	7	16	343
Raven	4	2	6	128
Laptop	0	6	6	132
Web Pad	0	6	6	132
All Nodes	34	34	68	1,462

Table 8 Configuration overhead for plentiful power composition with semantics

	wire_to events	wire_from events	Total events	Total bytes	Effic. gain (%)
SPOT	5	1	6	141	52
Raven	2	0	2	44	48
Laptop	0	8	8	188	57
Web Pad	0	8	8	188	57
All Nodes	18	18	36	834	54

Table 9 Configuration overhead for low power composition with semantics

	wire_to events	wire_from events	Total events	Total bytes	Effic. gain (%)
SPOT	5	3	8	189	45
Raven	3	1	4	91	29
Laptop	0	6	6	144	−9
Web Pad	0	6	6	144	−9
All Nodes	22	22	44	1,030	30

compositions, respectively, without subtyping (i.e., as in the original version of LooCI without a semantic type system). Tables 8 and 9 show the number of events that are required with subtyping. Without subtyping, the Event parameter in the *wireTo* and *wireFrom* commands (see Listing 1) is simply the event's SHORT_ID. With subtyping, the Event parameter is composed of both the event's SHORT_ID and GUID. The *Total bytes* column in the tables presents the summed size of the payloads of the wire events needed.

As can be seen from Tables 6, 7, 8 and 9, in the case of the plentiful power composition, subtyping reduces the bandwidth overhead of configuration from 1,806 bytes to 834 bytes, a 54 % increase in efficiency. In the case of the scarce power composition, the use of semantic component interfaces reduces the bandwidth overhead of configuration from 1,462 bytes to 1,030 bytes, a 30 % increase in efficiency. Note that the negative gain in efficiency for the Laptop and Web Pad in Table 9 are caused by the fact that the same number of events are required for their configuration, yet additional bytes are needed to represent the GUIDs when the semantic type system is used.

In addition to reducing bandwidth overhead, the use of subtyping also reduces configuration effort; as each wiring event requires one command to generate, subtyping reduces the total number of commands required to enact the plentiful power composition from 84 commands to 36 and from 68 commands to 44 for the low power composition. Similar savings of configuration effort should be expected when writing common component discovery and reconfiguration code.

6 Related work

This section reviews important related work in the area of semantic description languages and embedded component models.

6.1 Semantic description languages

Semantic web technologies like RDF-S [21], OWL [22] and SSN [23] may be used to build formal ontologies that allow machines to process, and act upon, ontological information in an automated manner. They are a key component in the Semantic Web. The Resource Description Framework Schema (RDF-S) [21] is a language that is used to formally describe resources and the relationships between them, including subtyping as provided in our scheme. The Web Ontology Language (OWL) [22] builds on RDF, and adds additional vocabulary and support for a more complex relationships, providing greater expressiveness. Unlike RDF, efficient inference engines such as RACER [24] are available for the OWL-DL and OWL-Lite sub-languages. These description languages are key building blocks of the four well-known versions of Berners–Lees Semantic Web layered architecture to enhance Web Service (WS) descriptions from pure machine-readable specifications into semantic descriptions that are also understood by computer systems. The Simple Semantic Web Architecture and Protocol (SSWAP) [25], the Semantic Sensor Web (SSW) [26] and the SONGS [27] project are other initiatives that use semantic services as abstractions for software components that process data with clear semantic interpretations. The survey in [28] provides an overview of the state of the art for the semantic specification of sensors. The Semantic Sensor Network (SSN) ontology [23] builds on OWL-DL, to integrate the resources of sensor network deployments into the Semantic Web, both on a data and management level. Its intended use is outside of the sensor network rather than providing support for in-network operations like binding and discovery of services. As such, we believe our work is highly complementary.

With respect to our requirements for compatibility and subtype testing of interfaces, these technologies incur needlessly high networking and computational overhead. Their use of the XML syntax as most common serialization format causes even simple semantic descriptions to be of unacceptable size for in-network use in WSNs. While projects such as EWSA [29] and BinaryWS [30] have shown that XML-based data can be efficiently encoded, this potential is limited by the inherent complexity of open-world semantic description languages. This causes simple operations like subtype testing to incur high computational overhead compared to our approach. Consequentially, we have adopted a modified version of the scheme described in [10], which uses the unique properties of prime numbers to generate a more compact ontology representation that supports efficient compatibility testing. The EASY [31] discovery protocol also adopted the semantic matching scheme in [10] for semantic service discovery in pervasive computing environments, with its efficiency for resource-constrained devices being demonstrated in [32]. In sum, these technologies provide more logical constructs and expressiveness than we believe is required or can be feasibly supported in WSN environments.

6.2 Component models and middleware

In this section, we discuss related work on component models and middleware. We highlight method-based and message-oriented approaches for both traditional distributed systems and WSNs.

Bindings and interfaces are common elements in the specification of component models and are required to compose components together into (distributed) applications. In method-based component models, subtyping of interfaces is often provided in a manner closely related to the concept of inheritance in object-oriented programming. Two such examples are Fractal [33] and OpenCOM [2]. Fractal is a modular and programming language agnostic component model used to build various systems and applications such as operating systems, middleware and graphical user interfaces. It provides explicit support for reflection, allowing components to be introspected, e.g., during reconfiguration. Although comprehensive and suitable for large-scale applications, Fractal's minimal core is too large to be applied on resource-constrained WSN platforms.

The OpenCOM component model [2] is a generic, platform independent component model, which has been used in the WSN domain to specify the RUNES [3] component model and the Lorien operating system [5]. In addition to components, OpenCOM offers the notion of Component Frameworks, which can be used to constrain patterns of how compatible components [6] should be composed together. The RUNES model [3] has a smaller footprint and adds a number of introspection API calls to the OpenCOM kernel. Lorien, on the other hand, introduces support for node-local type safety in software composition [34] through the use of formal component interfaces annotated with compile time generated hash codes. The interfaces of two components can only be interconnected if their type name and hash codes match. Subtype testing and remote binding is not addressed.

The component models described above offer runtime 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 reuse third-party components on the fly. In addition, these component models provide no support for compatibility checking between interfaces, which increases the burden on the developer and the scope for errors in application composition.

With regard to message-oriented middleware, we focus on the publish–subscribe communication paradigm. Subtype testing and type safety are common properties of pub/sub-systems [35] on traditional platforms such as WebSphere MQ [36] and JMS [37]. They are typically run on resource-rich devices and provide centralized message brokers that manage subscription tables and apply a matching algorithm to determine message forwarding. These solutions however introduce communication and processing overhead that precludes their direct application to resource-constrained WSN platforms. LooCI draws inspiration from such systems in the design of the distributed Event Bus.

TinyCOPS [38] is a WSN component model that provides a content-based publish–subscribe service. It allows subscribers to express interest in data through a conjunction of constraints over attribute values. Only messages that contain data that satisfy the constraints in the subscription are forwarded to the respective subscribers. The overall application and communication properties of the system can also be influenced by the application designer by specifying orthogonal properties in the subscription such as the communication protocol to use for subscription and notification delivery, the sampling rate to be used or whether cached values are acceptable.

The Mires [39] middleware implements a topic-based publish/subscribe middleware for WSNs that reuses the component model provided by NesC [1]. It incorporates routing into the middleware by providing an interface to which multiple routing protocol implementations can be plugged in. As with TinyCOPS, it allows additional services such as aggregation to be incorporated, which control the dissemination of messages.

DSWare [40] is a real-time event detection service based on the publish/subscribe paradigm that allows users to be notified of the occurrence of phenomena. Phenomena are described as combinations of events that take place within a specified time interval and space. To enable this, DSWare utilizes an event hierarchy in which atomic events are grouped together into compound events. For instance, temperature, light and acoustic events may be grouped into an explosion event. The proposed event hierarchy thus groups together events based on their relevance, instead of their semantic meaning as in LooCI's event taxonomy.

Compared to LooCI and its semantic interfaces, TinyCOPS, Mires and DSWare follow a more data-centric approach, focusing more on the efficient dissemination of data in the WSN. This is realized by applying techniques like aggregation, efficient routing and translation of low-level events into higher-level ones. However, these systems do not define clear interfaces of publishers and subscribers that can be introspected. As such, no support is provided to third parties to discover compatible components and reconfigure their bindings.

7 Limitations and future work

While we believe that the work proposed in this paper represents a significant contribution to the state of the art in component models for WSNs, we also recognize that the proposed approach has a number of limitations in its current form:

Evolution of the Type System In the current scheme, extensions to the type system propagate to the network as required by deployed components, and unused leaf types may be pruned so long as they are not being used by a deployed component. However, optimization or restructuring of the type system requires a complete middleware update and is not backwards compatible. We are currently exploring mechanisms for propagating these updates from the back-end to the WSN that do not necessitate wholesale reinstallation of the middleware.

Limits of Nominative Typing The proposed scheme is nominative in the sense that it allows for equivalence and subtype testing of explicitly declared service types. This offers limited scope for the construction of generic functional components: For example, consider a generic *averager* component that is capable of averaging any data payload encoded as an integer. Various types encoded this way may be scattered throughout a nominative taxonomy; however, the type system provides no mechanism to model structural equivalence of types. To address this, we are now investigating the separation of the single type system proposed here into a nominative type system for services and a structural type system for data payload description, to combine the advantages of both schemes.

Our current work also does not exploit the rich field of research in semantic reasoning. We are currently investigating how such work could be applied to support interaction between divergent WSN type systems.

8 Conclusions

This paper argues that to better support component discovery and reconfiguration, component interfaces should become self-describing, providing a precise semantic description of the service that is offered or required. In order to support this vision, we have created a semantic type system for component interfaces that is based on the embedded ontology description language presented in [10]. Worst-case analytic evaluation of this scheme indicates that it has minimal overhead even for large type systems.

The proposed semantic type system has been integrated with the LooCI component model [4] and evaluated in a small-scale WSN scenario. Our evaluation shows that the scheme incurs minimal overhead, with acceptable performance even on embedded WSN hardware such as the Raven mote [12]. In addition to reducing the scope for errors in application composition by enforcing type safety, this scheme has been shown to reduce application configuration overhead and to reduce message-passing overhead during application configuration.

Acknowledgments This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund KU Leuven and the XJTLU Research Development Fund and conducted in the context of the TEPAWSN project, the IWT-SBO-STADIUM project [41] and the IWT-SBO-SymbioNets project [42].

References

1. Gay D, Levis P, Von Behren R, Welsh M, Brewer E, Culler D (2003) The NesC language: a holistic approach to networked embedded systems. In: Proceedings of programming language design and implementation (SIGPLAN03), San Diego, California, USA, June 2003, p 111
2. Coulson G, Blair G, Grace P, Taiani F, Joolia A, Lee K, Ueyama J, Sivaharan T (2008) A generic component model for building systems software. ACM Trans Comput Syst 26(1):1–42
3. Costa P, Coulson G, Gold R, Lad M, Mascolo C, Mottola L, Picco GP, Sivaharan T, Weerasinghe N, Zachariadis S (2007) The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: Proceedings of the 5th annual IEEE international conference on pervasive computing (PerCom07), White Plains, New York, Mar 2007, p 6978
4. Hughes D, Thoelen K, Horré W, Matthys N, Michiels S, Huygens C, Joosen W, Ueyama J (2012) Building wireless sensor network applications with LooCI. Int J Mob Comput Multimedia Commun 2(4):38–64

5. Porter B, Coulson G (2009) Lorien: a pure dynamic component-based operating system for wireless sensor networks. In: Proceedings of the 4th international workshop on middleware tools, services and run-time support for sensor networks (MidSens'09), Urbana Champaign, Illinois, USA, Dec 2009, pp 7–12
6. Hughes D, Greenwood P, Coulson G, Blair G, Pappenberger F, Smith P, Beven K (2007) An experiment with reflective middleware to support grid-based flood monitoring. Concurr Comput: Pract Exp 20(11):1303–1316
7. Grace P, Hughes DR, Porter B, Blair GS, Coulson G, Taiani F (2008) Experiences with open overlays: a middleware approach to network heterogeneity. In: Proceedings of the European conference on computer systems (EuroSys08), Glasgow, Scotland, UK, Apr 2008, pp 123–136
8. Langendoen K, Baggio A, Visser O (2006) Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In: Proceedings of the 20th international conference on parallel and distributed processing (IPDPS'06), Washington, DC, USA, Apr 2006, pp 174–174
9. Schaffers H, Komninos N, Pallot M, Trousse B, Nilsson M, Oliveira A (2011) The future Internet. Domingue J, Galis A, Gavras A, Zahariadis T, Lambert D (eds) Smart cities and the future Internet: towards cooperation frameworks for open innovation. Springer, Berlin , pp 431–446
10. Preuveneers D, Berbers Y (2008) Encoding semantic awareness in resource-constrained devices. IEEE Intell Syst 23(2):26–33, 1541–1672
11. Thoelen K, Matthys N, Horré W, Huygens C, Joosen W, Hughes D, Fang L, Guan S (2010) Supporting reconfiguration and re-use through self-describing component interfaces. In: Proceedings of international workshop on middleware for sensor networks (MidSens10), Bangalore, India, Nov 29th–3rd Dec 2010, pp 29–34
12. Avr RZ Raven Data Sheet (2012) http://www.atmel.com/dyn/resources/prod_documents/doc7911.pdf. Accessed 4 May 2012
13. Dunkels A, Grönvall B, Voigt T (2004) Contiki: a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of 29th international conference on local computer networks (LCN04), Tampa, FL, USA, Nov 2004, pp 455–462
14. Simon D, Cifuentes C, Cleal D, Daniels J, White D (2006) Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In: Proceedings of the 2nd international conference on virtual execution environments, Ottawa, Canada, June 2006, pp 78–88
15. SUN SPOT Theory of Operation (2012) http://www.sunspotworld.com/docs/Yellow/SunSPOT-TheoryOfOperation.pdf. Accessed 4 Mar 2012
16. OSGi Alliance (2007) About the OSGi service platform. Technical whitepaper, revision 4.1, June 2007
17. The LooCI Project on Google Code (2012) http://code.google.com/p/looci/. Accessed 4 Apr 2012
18. Matthys N, Afzal RS, Huygens C, Hughes D, Michiels S, Joosen W (2010) Towards fine-grained and application-centric access control for wireless sensor networks. In: Proceedings of 25th symposium on applied computing, Sierra, Switzerland, Mar 2010, pp 793–794
19. Götz T, Suhre O (2004) Design and implementation of the UIMA common analysis system. IBM Syst J 43(3):476–490
20. Henninger S (1996) Supporting the construction and evolution of component repositories. In: Proceedings of the 18th international conference on software engineering (ICSE96), Berlin, Germany, pp 279–288
21. RDF Semantic Web Standard (2012) W3C standard. http://www.w3.org/RDF/. Accessed 4 Apr 2012
22. OWL Web Ontology Language Overview (2012) W3C recommendation. http://www.w3.org/TR/owl-features/. Accessed 4 Apr 2012
23. Semantic Sensor Network (2012) (W3C) incubator group charter. http://www.w3.org/2005/Incubator/ssn/charter. Accessed 4 Apr 2012
24. Haarslev V, Möller R (2003) Racer: a core inference engine for the semantic web. In: Proceedings of 2nd international workshop on evaluation of ontology-based tools (EON2003), Sanibel Island, Florida, USA, Oct 2003, pp 27–36
25. Gessler DDG, Schiltz GS, May GD, Avraham S, Town CD, Grant D, Nelson RT (2009) SSWAP: a simple semantic web architecture and protocol for semantic web services. BMC Bioinformatics 10:309
26. Sheth A, Henson C, Sahoo S (2008) Semantic sensor web. In: IEEE Internet computing, July 2008, pp 78–83
27. Liu J, Zhao F (2008) Composing semantic services in open sensor-rich environments. IEEE Network 22(4):44–49
28. Compton M, Henson C, Lefort L, Neuhaus H, Sheth A (2009) A survey of the semantic specification of sensors. In: Taylor K, Ayyagari A, De Roure D (eds) Proceedings of the 2nd international workshop on semantic sensor, networks, pp 17–32
29. Abangar H, Ghader M, Gluhak A, Tafazolli R (2010) Improving the performance of web services in wireless sensor networks. In: Proceedings of future network and mobile summit, Florence, Italy, July 2010, pp 1–8
30. Castellani AP, Ashraf MI, Shelby Z, Luimula M, Yli-Hemminki J, Bui N (2010) BinaryWS: enabling the embedded web. In: Proceedings of future network and mobile summit, Florence, Italy, July 2010, pp 1–8
31. Ben-Mokhtar S, Preuveneers D, Georgantas N, Issarny V, Berbers Y (2007) EASY: efficient semantic service discoverY in pervasive computing environments with QoS and context support. J Syst Softw 81(5):785–808
32. Mokhtar S, Raverdy PG, Urbieta A, Cardoso RS (2010) Interoperable semantic and syntactic service discovery for ambient computing environments. Int J Ambient Comput Intell 2(4):13–32
33. Bruneton E, Coupaye T, Leclercq M, Quma V, Stefani J-B (2006) The FRACTAL component model and its support in Java. Softw Pract Exp 36(11–12):1257–1284
34. Porter B, Roedig U, Coulson G (2011) Type-safe updating for modular WSN software. In: Proceedings of the 7th IEEE international conference on distributed computing in sensor systems (DCOSS11), Barcelona, Spain
35. Eugster P, Felber P, Guerraoui R, Kermarrec A-M (2003) The many faces of publish/subscribe. ACM Comput Surv 2:35
36. IBM Corporation (1995) MQSeries: an introduction to messaging and queuing. Technical report GC33-0805-01. IBM Corporation, Yorktown Heights
37. Hapner M, Burridge R, Sharma R, Fialli J, Stout K (2002) Java Message Service. Sun Microsystems Inc., Santa Clara
38. Hauer J, Handziski V, Kopke A, Willig A (2008) A component framework for content-based publish/subscribe in sensor networks. In: Proceedings of 5th European conference on sensor networks (EWSN08), Bologna, Italy, Jan 2008, pp 369–385
39. Souto E, Guimarães G, Vasconcelos G, Vieira M, Rosa N, Ferraz C, Kelner J (2005) Mires: a publish/subscribe middleware for sensor networks. Personal Ubiquitous Comput 10(1):37–44
40. Li S, Lin Y, Son SH, Stankovic JA, Wei Y (2004) Event detection services using data service middleware in distributed sensor networks. Telecommun Syst 26(2):502–517
41. IWT-SBO-Stadium project No. 80037 (2009) Software technology for adaptable distributed middleware. http://distrinet.cs.kuleuven.be/projects/stadium/
42. IWT-SBO-SymbioNets project No. 090062. Symbiotic networks. http://symbionets.intec.ugent.be