Composition-safe re-parametrization in Distributed Component-based WSN Applications

Wilfried Daniels*, Pedro Javier del Cid*, Danny Hughes*, Sam Michiels*, Chris Blondia[†], Wouter Joosen*

*iMinds-DistriNet, KU Leuven, Heverlee, B-3001, Belgium

Email: {firstname.lastname}@cs.kuleuven.be

[†]iMinds, Universiteit Antwerpen, Antwerp, B-2000, Belgium.

Email: Chris.Blondia@ua.ac.be

Abstract—Contemporary Wireless Sensor Networks like Smart Offices and Smart Cities are evolving to become multi-purpose application hosting platforms. These WSN platforms can simultaneously support multiple applications which may be managed by multiple actors. Reconfigurable component models have been shown to be viable solutions to reducing the complexity of managing and developing these applications while promoting software re-use. However, implicit parameter dependencies between components make reconfiguration complex and error-prone. Our approach achieves automatic composition-safe re-parametrization of distributed component compositions. To achieve this, we propose the use of language annotations that allow component developers to make these dependencies explicit and constraint-aware network protocols to ensure constraint propagation and enforcement.

I. INTRODUCTION

In recent years, Wireless Sensor Network (WSN) infrastructures have changed radically. Earlier WSNs were geared towards single purpose applications with little or no runtime reconfiguration support. Typically, a single party built, owned and maintained the WSN. With the advent of shared sensing infrastructures, e.g. Smart Cities [1] and Smart Offices [2], WSN infrastructures are evolving to become more open and multi-purpose [3], [4]. A single WSN infrastructure can host multiple applications at the same time that are managed by multiple actors.

Reconfigurable component models have proven to be a promising solution for these new challenges [4]. Examples of modern runtime reconfigurable component systems include OpenCOM [5], RUNES [6], OSGi [7], REMORA [8] and LooCI [9]. These systems provide the capabilities required to manage component life-cycle, configuration, introspection, and assembly at runtime. In component-based systems, functionality is encapsulated in software artifacts with clearly defined interfaces. Components are interconnected or ‘composed’ into distributed applications or ‘compositions’. Compositions may be shared between multiple applications. Component functionality is parametrized with the use of configuration properties, e.g. sampling interval in a component that provides access to a temperature sensor, or configuring the window size of an aggregating component.

An essential feature of component-based system in multi-actor multi-purpose WSN infrastructures is component re-use.

Application developers build their application by composing previously deployed and new components. This way, a single running instance of a component on a WSN platform can provide functionality in multiple component compositions, sharing both platform resources (i.e. Flash, RAM) and code.

Contemporary component-based systems have some drawbacks when sharing component instances in multiple compositions. Configuration conflicts may arise due to resource competition and contention. While component binding dependencies are explicit in the form of interfaces and receptacles, implicit dependencies may still exist in a software composition due to application level constraints which are enforced through the setting of configuration parameters. Consider a software composition that detects vehicle motion, using a composition of a magnetometer component and a motion detection component. The magnetometer component must sample at 16 Hz in order for the motion detection component to function properly. This required configuration generates an implicit *parameter* dependency between the magnetometer and motion detection components that is not currently expressible with state of the art component models and their associated language constructs.

Resolving these implicit dependencies is difficult and error-prone when done manually. In a multi-purpose WSN infrastructure where multiple actors re-use and reconfigure components, no single actor has an accurate understanding of all existing compositions and parameter dependencies. When reusing and re-parametrizing a pre-existing component in a new application, existing compositions have to be introspected remotely, which incurs developer overhead and message passing overhead. Failure to resolve an implicit parameter dependency will cause disruption due to erroneous configurations of existing applications.

In this paper we present an approach that externalizes implicit distributed dependencies generated from component parametrization in distributed applications. This allows for automatic composition-safe re-parametrization in multi-actor WSNs. Our solution has 2 elements: Firstly, we provide language constructs which can be used by component developers to make parameter constraints and dependencies explicit across component compositions. Secondly, a network protocol is presented which automatically resolves these dependencies

when composing components at runtime and flags constrained parameters with the corresponding constraints.

II. MOTIVATION

Over nine months ago our research group created a smart office environment in our facility. Conceived and designed to be a reusable sensing infrastructure it started out supporting one application and quickly evolved to support four. These applications are: facility management, workforce management, security and workplace safety, each of which is managed by a different stakeholder. We have continuously monitored reconfiguration actions and logged reconfiguration effort and latency. Our analysis of the main causes of reconfiguration effort and latency revealed, for the first time, the problem of implicit dependencies between parametrized components in a composition. In order to further investigate, we conducted a series of experiments designed to better understand the impact that implicit dependencies have on reconfiguration effort and latency.

1) *Experiment description:* During the experiments, we tasked seven experienced component developers with a series of reconfiguration exercises to be conducted on the smart office. In each exercise the component assembler had to plan and enact an extension to one of the running applications. Throughout the experiment we tracked: reconfiguration effort, latency and disruption.

We assumed there is no inter-stakeholder coordination of reconfiguration plans and there is no up to date global view of the system. The assemblers all started out having limited information in regards with the configuration and state of all the running component instances and the applications.

2) *Results:* Analysis revealed a significant problem not identified by the literature. We ascertained that implicit dependencies between components in composition exist every time the consumer component has application requirements that constrain the possible values used to configure the functionality implemented by the producer component. Further analysis reveals that this is a generic problem for all multi-user component based systems.

An example from the smart office environment is given in Figure 1, where a component responsible for managing climate control requires the temperature sensing component to sample at a fixed sampling rate of 12 Hz. This dependency spans over a component composition and can cross node boundaries.

Dependencies such as the one described incur a management overhead in a multi-user environment. If an other user wants to re-use the *Temp Sensor* component in a different composition which requires a different sample rate, the entire composition has to be checked for possible implicit dependencies which would prevent altering the sampling rate. Resolving this dependency manually would require introspection of all components on all nodes involved, resulting in a potentially high management overhead.

III. BACKGROUND

In this section we discuss the requirements of our approach on component based middleware. We then enumerate the

Fig. 1. Implicit dependency over a component composition

roles that a component can play during composition-safe re-parametrization, and finally provide a classification of constraint types.

A. Component model requirements

In order for our approach to be applicable, the runtime re-configurable component model should offer: (i) explicit interface and receptacle declarations, (ii) a **unique identifier** (*uid*) for each interface and receptacle type, (iii) re-parametrization of running components and (iv) binding (i.e. interconnection) of components should only be allowed where interfaces and receptacles are of the same type. To the best of our knowledge, these requirements are met by all runtime reconfigurable components models, including: RUNES [6], REMORA [8] and LooCI [9]. Figure 2a shows an example component compositions with *uid*-annotated interfaces and receptacles.

B. Component roles

After analysing the nature of parameter dependencies in the smart office environment, three component roles were identified, each of which must be considered when resolving distributed parametrization dependencies. We describe each of these roles, with reference to the example smart-office composition shown in Figure 1.

- 1) **Constrained components:** These are components which produce events differently based upon their parametrization. A concrete example of such a component is the *Temp Sensor* component shown in Figure 1. The Sampling Rate parameter (SR) influences how often temperature data is sensed and transmitted, which is constrained by the *Climate Control* component.
- 2) **Relaying components:** Relay components do not have constrained parameters, or constrain the parametrization of other components. They do however serve as a relay of parameter constraints along the chain of components in a composition. In Figure 1, the *°C to °F converter* is an example of a relaying component. It relays data between a constrained component *Temp Sensor* and the constraining component *Climate Control*.
- 3) **Constraining components:** Constraining components consume data that is produced and processed by other components in the composition. Constraining components require a specific parametrization of components producing and processing the data. The *Climate Control* component shown in Figure 1 is an example of a constraining component. It requires that the sampling rate of the *Temp Sensor* component has a fixed value. Another possibility is constraining the range of acceptable parametrization, e.g. 10 to 14 Hz.

These roles are not mutually exclusive. A component may play multiple roles at the same time.

IV. DESIGN

A. Language annotations

In order to offer a composition-safe component model, component developers must specify parameter dependencies, relaying behaviour and constraints. To achieve this, we introduce a simple and consistent set of language annotations. These annotations are used differently by each component according to its role (constrained, relaying or constraining) as identified in the previous section.

For *constrained components*, developers use the syntax shown in Listing 1 to identify constrained parameters. Constrained parameters are assigned an id, which is used by the constraint resolution protocol to reference the component. The component developer must specify both the id of the constrained parameter and the *uid* of the outgoing interface. Figure 2a shows an annotated version of the composition visualized in Figure 1. In this scenario, the *Temp. Sensor* component specifies `ConstrainedParameter(sr_id, uid1)` to define the constrained parameter.

For *relaying components*, developers must specify the *uid* of the incoming interface and the outgoing receptacle over which the dependencies have to be relayed. Listing 2 shows the syntax that is used by relaying components. For example, the $^{\circ}C$ to $^{\circ}F$ component in Figure 2a specifies `DependencyRelay(uid1, uid2)`.

For *constraining components*, developers use the syntax shown in Listing 3. Constrained parameters are designated by their parameter id together with the *uid* of the incoming receptacle. The tuple formed by this pair of values uniquely specifies a specific constrained parameter. Constraints are specified by appending the parameter identifying tuple with the constraint itself, which is either a valid range or a fixed single value. The example constraint of the *Climate Control* component shown in Figure 2a is expressed as follows `ParameterLock(sr_id, uid2, 16Hz)`.

For all component roles, it should be noted that language annotations are compact and require little additional effort from the component developer.

```
ConstrainedParameter(
    parameter-id,    //Reference to constr. parameter
    interface-uid);  //Outgoing interface uid
```

Listing 1. Language annotations defining constrained parameters

```
DependencyRelay(
    receptacle-uid, //Incoming receptacle uid
    interface-uid); //Outgoing interface uid
```

Listing 2. Language annotations defining dependency relaying

```
ParameterLock(
    parameter-id,    //Reference to constr. parameter
    receptacle-uid, //Incoming receptacle uid
    constraint);     //Open/closed interval or value
```

Listing 3. Language annotations defining parameter constraints

B. Constraint propagation protocol

The language annotations described in the previous section make parameter dependencies explicit. In this section, we introduce a network protocol that efficiently relays appropriate constraints from constraining components to constrained components. It is assumed components are initially unconnected and that applications are constructed at runtime by connecting the compatible interfaces and receptacles of components. It should be noted that bindings may be local, or may cross node boundaries, requiring the transmission of a radio message.

The constraint propagation protocol works in two phases. In *phase one*, descriptions of constrained parameters are propagated along the chain of components as bindings are made. A caching mechanism is used at every component along the chain from the constrained component to the constraining component. Every time a new binding is made, the local cache of the component is checked for constrained parameters that should be forwarded. These cache entries are then forwarded along the chain as far as existing bindings allow. Passing constraint information along the component graph is required because components have no a-priori knowledge of the application composition in which they will participate. In *phase two*, constraints are sent back directly to the constrained components.

The process of constraint propagation for the composition shown in Figure 2a is shown in Figure 2b. When the first binding is made, *Temp. Sensor* forwards its constrained parameter to the $^{\circ}C$ to $^{\circ}F$ component, which caches it. Next, when the last binding is made, $^{\circ}C$ to $^{\circ}F$ forwards this constrained parameter to *Climate Control*, which matches it with a lock constraint and sends this constraint back directly.

It should be noted, that, our approach enforces no ordering constraints on application composition and communication is minimal, occurring only when an explicit binding is made or a parameter is adjusted.

The result of the constraint propagation protocol discussed in the previous section is the local storage of constraint specifications at all relevant points in the component composition. These locally stored parameter constraints are enforced during re-parametrization.

V. RELATED WORK

In this section we provide an overview of the state of the art in reconfiguration approaches used in component-based systems. Reconfiguration approaches in component-based systems for WSNs can be broadly categorized into structural and behavioral reconfiguration [10]. Structural reconfiguration is defined by the ability to modify the compositional topology. Behavioral reconfiguration on the other hand allows for the modification of component behavior by offering fine-grained adjustments at runtime.

1) *Structural reconfiguration*: In the context of WSNs, several well known component-based systems support structural reconfiguration. Examples include RUNES [6], REMORA [8] and LooCI [9]. In these systems, components can be deployed, configured and interconnected at runtime. Our approach builds

Fig. 2. Component annotations and constraint resolution

on top of and is complementary to these approaches by extending their capabilities through the externalization of implicit parameter dependencies.

2) *Behavioral reconfiguration*: Fine grained reconfiguration is commonly achieved with the modification/insertion of software modules smaller than components or by modifying component configuration parameters. Smaller software modules are commonly offered as policies or micro-components. The former allow for the insertion of functionality implemented in policies, redirection of data and control flows as proposed in PMA by Matthys et al. [11]. The latter allow for the modification of functionality within components, as proposed in ReWise micro-components by Taherkordi et al. [12]. In ReWise a regular component can isolate small pieces of functionality in TinyComponents, which are runtime replaceable. Modifying component configuration parameters is commonly achieved by exposing a configuration interface, as in LooCI [9], RUNES [6] and REMORA [8].

VI. CONCLUSION AND FUTURE WORK

In this paper we identified for the first time a new problem for distributed component-based systems, more specifically implicit parameter dependencies. These implicit distributed dependencies occur when parameters are constrained across component compositions and introduce complexity when re-parametrizing.

We propose a solution which leverages annotated components to externalize and enforce these implicit constraints at runtime. By resolving parameter constraints in the middleware, our approach greatly simplifies re-parametrization by avoiding the need for extensive introspection to identify these implicit dependencies.

Possible future work includes extending the proposed solution with constraints that enforce synchronization of parameters across compositions. Furthermore, we aim to implement our approach on an existing component-based platform in order to validate feasibility and assess gains in re-parametrization effort along with possible overheads.

ACKNOWLEDGEMENTS

This research was partially funded by the Research Fund KU Leuven, IWT and iMinds, and conducted in the context

of the IWT-SBO-SymbioNets project (IWT 90062).

REFERENCES

- [1] L. Filippini, A. Vitaletti, G. Landi, V. Memeo, G. Laura, and P. Pucci, "Smart City: An Event Driven Architecture for Monitoring Public Spaces with Heterogeneous Sensors," in *Conference on Sensor Technologies and Applications (SENSORCOMM '10)*. IEEE, Jul. 2010, pp. 281–286.
- [2] S.-y. Lau, T.-h. Chang, S.-y. Hu, H.-j. Huang, L.-d. Shyu, C.-m. Chiu, and P. Huang, "Sensor Networks for Everyday Use: The BL-Live Experience," in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06)*. IEEE, 2006, pp. 336–343.
- [3] A. Rezgui and M. Eltoweissy, "Service-oriented sensoractuator networks: Promises, challenges, and the road ahead," *Computer Communications*, vol. 30, no. 13, pp. 2627–2648, Sep. 2007.
- [4] C. Huygens, D. Hughes, B. Lagaisse, and W. Joosen, "Streamlining Development for Networked Embedded Systems Using Multiple Paradigms," *IEEE Software*, vol. 27, no. 5, pp. 45–52, Sep. 2010.
- [5] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, vol. 26, no. 1, pp. 1–42, Feb. 2008.
- [6] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, "The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario," in *IEEE International Conference on Pervasive Computing and Communications (PerCom'07)*. IEEE, 2007, pp. 69–78.
- [7] J. S. Rellermeyer and G. Alonso, "Concierge: A Service Platform for Resource-Constrained Devices," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, p. 245, Jun. 2007.
- [8] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "Programming sensor networks using REMORA component model," in *IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '10)*, 2010, pp. 45–62.
- [9] D. Hughes, K. Thoenen, J. Maerien, N. Matthys, P. J. del Cid Garcia, W. Horré, C. Huygens, S. Michiels, and W. Joosen, "LooCI: the Loosely-coupled Component Infrastructure," in *IEEE International Symposium on Network Computing and Applications (NCA '12)*, 2012.
- [10] M. Aksit and Z. Choukair, "Dynamic, adaptive and reconfigurable systems overview and prospective vision," in *International Conference on Distributed Computing Systems - Workshops (ICDCS '03)*. IEEE, 2003, pp. 84–89.
- [11] N. Matthys, C. Huygens, D. Hughes, S. Michiels, and W. Joosen, "A Component and Policy-Based Approach for Efficient Sensor Network Reconfiguration," in *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY '12)*. IEEE, Jul. 2012, pp. 53–60.
- [12] A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "Supporting lightweight adaptations in context-aware wireless sensor networks," in *International Workshop on Context-Aware Middleware and Services (CAMS '09)*. New York, New York, USA: ACM Press, 2009, p. 43.