Towards component orientation in embedded web service environments

Elmar Zeeb, Guido Moritz, Dirk Timmermann
Institute of Applied Microelectronics
and Computer Engineering
University of Rostock
18057 Rostock, Germany
{elmar.zeeb, guido.moritz, dirk.timmermann}
@uni-rostock.de

Frank Golatowski
Center for Life Science and Automation
18119 Rostock, Germany
frank.golatowski@celisca.de

Abstract

Service-oriented architectures and systems designs are powerful concerning reusability of functional blocks and hiding implementation details from functional interfaces. But to compose a complete application, often central entities and engines are required for processing a specific sequence of service. In component-based designs the component itself is capable of describing both not only their offered services and interfaces but also dependencies on other services and interfaces to fulfill a complete task or application logic. This paper investigates on how to transfer and enhance exiting component-based approaches, already known from business applications, into the domain of embedded web services environments. Special focus is on the Devices Profile for Web Services (DPWS) technology which features service orientation also in device centric applications. This paper introduces a new approach to create applications, based on services provided by devices deployed with DPWS, in an abstract and dynamic way.

1. Introduction

Service-oriented technologies play an emerging role in embedded device centric scenarios in recent times. Existing industry standards for wired or wireless machine to machine (M2M) communication do not offer the flexibility required for future applications. Often protocols, concepts and the underlying physical technology are interwoven in each other. For applications that base on several different M2M technologies, high implementation efforts are required when concepts and protocols of technologies cannot be easily converged. Thus new technologies, protocols and concepts are required that overcome this shortcoming, while these solutions have to be flexible enough to allow legacy support for existing systems also.

Service-oriented Architectures (SOAs) are based on basic principles that try to keep application as flexible as possible. These principles can be leveraged in applications as well as in the enterprise system domain, where the service orientation originates from. Its principles like loose coupling and late binding are crucial factors for the success of service-oriented architectures in device centric applications. Especially the concept of the service that is the fundamental element of an application design fits into the device domain where devices offer distinct functionalities as remote services. Higher level services or applications thereby are composed of services in a way to minimize the dependencies between services and thus keep applications flexible.

Several technologies are available to implement such applications today. W3C Web Services are a widespread technology that addresses large scale enterprise systems primarily. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network [15]. Other technologies like Universal Plug and Play (UPnP) and OSGi Service Platform are addressing device scenarios directly. But with the standardization of the DPWS at the Organization for the Advancement of Structured Information Standards (OASIS), a consistent and W3C Web Service based technology for device centric scenarios is also available.

A basic service-oriented principle that is important in device centric applications is the composability. In SOAs basic functions of devices are modeled as services. Applications can be built by composing several device services. There is a tendency to create this composition in an abstract declarative or procudural way to ease changes and adaptations of the application. In the W3C Web Services domain compositions are often realized by centralized process-oriented solutions. Other technologies offer component-oriented solutions to compose service and enables direct interconnection of service consumer (client) and service provider (service). In contrast to a service a component not only offers services, but also can express relations and depenencies to other services which are required to provide a distinct functionality.

This paper focuses on how to create applications, based on services provided by devices deployed with DPWS, in an abstract and dynamic way. The aim is to avoid service composition directly in program code - what is assumed to be the most inflexible way of service composition.

In the following section 2, a more detailed description of DPWS is provided. In section 3, an overview of service composition approaches based on web services and how they can be used in device centric applications is given. In addition, other non web service based technologies for service composition that can be used in device centric scenarios are described. In the section 4, a new device centric and Web Services based composition approach with a special focus on DPWS is presented. Section 5 describes the implementation of the new approach with the WS4D-gSOAP DPWS toolkit. Finally, the new approach is discussed in section 6 and the conclusion in section 7 wraps up the results and benefits of the new approach.

2. DPWS and service composition

The increasing complexity of infrastructures and networks consisting of up to thousands of devices demands new technologies for simple device interaction and interoperability. SOAs [9] firstly addressed this issue for software components. The probably most widespread implementation of SOA are W3C Web services. For device to device communication, the Web services protocols often need too much resources and computing power, especially if resource constraints are implied due to deployment environment, application scenarios or hardware costs. The Web services technology also lacks features like ad-hoc device discovery, device description and eventing channels that are required in device networks. Thus, DPWS [13] was defined. It uses some specific Web services protocols and restricts their usage because of resource limitations in embedded systems. Furthermore DPWS includes enhancements to fit into device centric applications and offer required functionalities like e.g. the former mentioned discovery and eventing capabilities. So DPWS enables the usage of Web services based technologies to implement device centric SOAs and thus offers the same modular and clearly defined software architectures in device networks ([7], [11]).

DPWS is based on well known protocols and Web service specifications. It uses similar messaging mechanisms like the Web Services Architecture (WSA) [15] with restrictions to complexity and message size ([13], [16]). On top of the low level communication basics it uses Extensible Markup Language (XML), SOAP and XML-Schema for data and information exchange. DPWS specifies mechanisms for ad-hoc device discovery are based on WS-Discovery. The device and service descriptions are based on WS-MetadataExchange and WS-Transfer. Publish-subscribe mechanisms for push messaging in contrast of pull communication pattern are achieved by using WS-Eventing.

In general, DPWS is very similar to UPnP while the main difference is the direct alignment of DPWS to the latest Web Service specifications. In UPnP the referenced specifications are not updated.

DPWS defines a client role that uses the features that are described in the following paragraphs and a device role that implements these features. The term DPWS peer used in this paper denotes a DPWS device with client functionality.

The discovery mechanisms enables devices to announce their availability in the local network with Internet Protocol (IP) multicast messages. Clients can listen for this messages or send messages to search for devices in the network. All discovery messages can contain device type and device scope information. A device type is a unique identifier that classifies a device defined at design time. For example printer device or scanner device are device types. The meaning of device types should be specified at design time in separate specifications comparable to a UPnP device control protocol (DCP). A device can support (implement) several device types. In contrast to device types, a device scope is a classification that can be configured at runtime. Room 1227 would be an example for a scope to identify all devices that are located in room 1227.

The description mechanism of DPWS enables the dynamic description of device metadata such as hosted services, device information, model information or service description. This metadata is associated with a metadata version number that is distributed in discovery messages: Thereby clients can track changes of the device description. The interfaces to retrieve the device description are based on WS-Metadataexchange. As specified in WS-Metadataexchange, metadata is partitioned into sections. DPWS defines which endpoints of an device should provide at least which metadata sections. Custom metadata sections can be specified to extend the device description data for custom applications.

DPWS devices can offer services with operations and events. Operations are the same as SOAP Web Services operations, whereby the services hosted on a DPWS device are regular SOAP Web Services. Events are controlled with WS-Eventing and represented as inverse SOAP Web Service operations. This means that client and service exchange roles and the message exchange is triggered by the device. To subscribe for an event, a client can send a subscribe message to the service endpoint. The subscribe message contains the requested delivery mode and event filter. With the delivery mode mechanism a client can negotiate a suitable delivery mechanism. DPWS defines the delivery push mode that sends the events to an endpoint specified in the delivery mode. Further application specific delivery modes can be defined. A dedicated event filter specifies which events the device should send. DPWS defines the action filter. SOAP and WS-Addressing define actions (identifiers) for operations. Thus event filters can be applied and mapped directly to the corresponding inverse operations. Like the flexibility to define own delivery modes, also own event filters can be used to meet applications needs.

As a partner of the Web Services for Devices (WS4D) initiative, the University of Rostock has developed the WS4D-gSOAP and the WS4D-Axis2 toolkit, which include a DPWS protocol stack implementation and some software tools to create devices and clients. Other toolkits for different platforms and programming languages are available from WS4D [3], Service-Oriented Architecture for Devices (SOA4D) [2] or as integral part of Windows Vista, Windows 7 and the .Net Framework [1].

There are several approaches how to compose services of DPWS devices, whereby each solution has its specific advantages and disadvantages. Creating the composition by using a regular programming language offers only a minimum of flexibility and extensibility. In the next section several approaches are discussed how DPWS devices and their services can be composed in a more flexible way.

3. Service composition approaches

In general there are several concepts and technologies to compose services and devices. A complete discussion is out of scope of this paper. Because DPWS is based on Web Services, this paper discusses existing Web Service composition technologies and if they can be applied in DPWS bases applications. Futhermore, other SOA based technologies addressing device centric scenarios are presented and analyzed if the underlying concepts can be transferred to DPWS. The first option is based on the eventing feature of DPWS and thus includes no new concepts. The next options WS-BPEL and WS4D-PipesBox realize a process-oriented composition approach, that has the major drawback of routing all messages through a central process runtime that executes a process. The last options Service Component Architecture (SCA) and iPojo realize a component-oriented approach. While iPojo cannot be applied to DPWS directly, the general concepts can as presented herein.

3.1. Eventing

Eventing features in DPWS are possible through the included WS-Eventing [12] specification. WS-Eventing defines several roles that are part of the process of subscription for and event, receiving events and managing subscriptions. As shown in figure 1, the roles are client, event source, subscription manager and event sink.

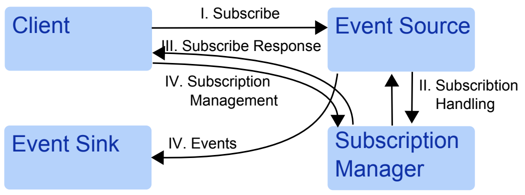


Figure 1. Roles in WS-Eventing

The client subscribes for an event source. This subscription is handled by the subscription manager that must not be necessarily the same entity like the event source. Because the event sink can be a different endpoint as the client, a third party can create a subscription. So in this case the third party can compose which event sink is connected to which event source.

This approach has two major disadvantages. On the one hand there are concepts missing how to describe and perform this composition in an abstract way. On the other hand only events are covered in this approach. Services that are not capable of eventing like typical controlling applications are excluded.

3.2. WS-BPEL

The Web Service Business Process Execution Language (WS-BPEL) standard is governed by OASIS [4]. It specifies an XML-based language to described processes of Web service interactions. In contrast to other process management standards, this specification also offers the direct execution of a process on an engine. In thevstrict sense, WS-BPEL is an imperative programming language for Web service interactions [6]. The current version WS-BPEL 2.0 is based on XML Schema for describing data structures, XPath for getting access to all elements in an XML structure, WSDL as self-describing interfaces to participating Web services as well as to the process itself, and WS-Addressing for the description of service endpoints

In [6], Bohn describes concepts for extending WS-BPEL with support for DPWS called BPEL4D. The major missing features of WS-BPEL for applying it to DPWS also are integration of WS-Discovery in general, the security concepts of DPWS and the publish/subscribe mechanisms included in DPWS. Especially the dynamic changing environment implied in device centric applications are not covered sufficient by WS-BPEL without extensions. But also the concept of differentiating between devices and services in DPWS cannot be mapped in WS-BPEL.

Another approach then extending WS-BPEL would be wrap the required features in a dedicated DPWS device and services and thus hide the required extensions from the WS-BPEL engine. This approach is out of this paper as described later in detail.

3.3. WS4D-PipesBox

In contrast to WS-BPEL that is fixed to describing W3C Web services processes, Bobek described in [5] an approach called WS4D-PipesBox following the Yahoo Pipes approach. PipesBox is focusing not only device or non-device centric composition scenarios, but is filling the gap to compose scenarios consisting of both devices and higher valued services. Thus the PipesBox tool is framework agnostic and features integration of heterogeneous technologies, devices and services in one application scenario. PipesBox can be applied in any domain, but main scopes are building automation, desktop applications and embedded devices.

To support the diverse technologies and protocols, PipesBox homogenizes technology specific capabilities in functionalities provided by the platform itself. In the lowest layer the specific libraries and tools are used to map into the corresponding protocols like DPWS, UPnP, Bluetooth, etc. On the next layer, modules perform specific tasks like discovery of devices or invocations of services and use the functionalities provided by lower layer libraries and tools. The upper layer is the application layer where no further specific technical knowledge is required and modules are connected with pipes by using a graphical user interface. Thus technical details are hidden from the designer of a pipe and modeling of functional blocks is much easier. Thereby also non technology specialists without programming knowlegde are able to compose a complete application.

3.4. Service Component Architecture

The Service Component Architecture (SCA)[1], standarized at OASIS, is a set of specifications to provide a programming model for building applications and systems based on a Service Oriented Architecture. In contrast to the formed mentioned solutions, SCA is a component-based framework. Business applications are often a sequence of different service invocations. These sequences can be combined to model the complete business need. Hence SCA related specifications define how to model such business scenarios and how to create business applications by either defining new components or reusing existing application functions.

To not rely on one specific technology, SCA describes different bindings for Web services, JMS and EJB. These binding should not be mixed up with SOAP binding known from W3C Web services, because the SCA bindings describe how to integrate the different technologies in their models. Additionaly SCA also defines how to describe components in different programming languages to enable heterogenous system design.

3.5. iPOJO

Like SCA described above, the iPOJO approach is also a component-based. The term *POJO* is derived from the phrase plain old Java object (POJO) and follows up the main goal of keeping development of service-oriented components (SOCs) as simple as possible [10]. To decouple the development of components and business logics from SOC mechanisms, a component container hides SOC specific aspects to ease implementation and realization of a complete scenario. After configuring the component container for application needs, developers only have to focus business logic.

Each container is described by its metadata which describes both offered services and required services. The container is also responsible for managing if all dependencies concerning required services at runtime by resolving missing or broken dependencies. The required binding on service specifications are not declared explicitly because they are already included in the metadata describing the dependencies. Technology specific binding details are managed by the container during runtime.

iPOJO is, like the WS4D-PipesBox approach, implemented on the service-oriented platform OSGi also, because OSGi already supports a sufficient numbers of platforms and thus can be applied in a large range of use cases.

4. DPWS based component-oriented approach

In this section a new concept and mechanism is introduced that features description of services a device can consume. This mechanism tries to be as close as possible to the available concepts and mechanisms in DPWS. In general, this approach is heading towards component orientation, but does not fully cover all issues that are solved in typical component-oriented frameworks.

The scope of this approach is to define an interoperable way to describe the services a DPWS peer can consume and a mechanism that enables third parties to control and configure which services a DPWS peer should use. In the next two sections the key parts of this approach are described. First the utilizable service metadata defines a language that is part of a devices metadata to express the consumers on a DPWS peer. Then the utilizable services manager is a service to assign specific endpoints to the consumers on a DPWS peer and thus control the binding of the consumers.

4.1. Utilized Services Metadata

DPWS defines a language to describe the services hosted on a device called Relationship. This language is used in the relationship section that is part of the metadata that can be retrieved with the description mechanism at run time. This language defines the host element that describes the device and the hosted element that describes a hosted service. The relationship can contain one host and several hosted elements. As the hosted element contains an endpoint reference (EPR), this is simply a list of EPRs of hosted services. There are further information contained in this element like the service type and the service ID. This enables clients to choose the right service EPR on devices which host several services. So the central element to describe hosted services is the hosted element in the corresponding relationship metadata section.

To describe service consumers, a new utilized element is introduced. In analogy to the hosted element, this element describes the services that can be used by a device. In contrast to the hosted element, it does not describe services but service consumers. A service consumer uses a service reference (address) to bind a specific service. So the utilized services element describes a service consumer on a device and to which kind of services it can bind to.

As DPWS has device types, service types and events, there are several ways to bind a service. Thus service con-

[1]http://www.oasis-opencsa.org/sca (2010)

sumers need different types of EPRs as input. There are at least three ways how to bind to a service in DPWS:

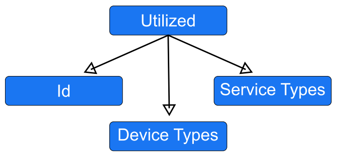
1. A service consumer can use devices of a specific device type and uses any services on this device. This consumer requires device EPRs which implement a specific device type as input. This binding allows highest possible abstraction in scenarios where devices types are related with hosted services directly.

2. A service consumer can use devices of a specific device type and uses services of a specific service type on this device. This consumer requires tuples of device and service EPRs that implement the specific device and service type as input.

3. A service consumer can use a regular web service of a specific type. This consumer requires a service EPR that implements the specific service type.

All further approaches how to bind a service can be derived from the combination of required device and service types. The binding information must be included in the utilized element. Device types correspond to device types as defined in the DPWS specification. Service types are port types as defined in Web Service Description Language (WSDL) port types. Additionally the service consumer can be identified with an ID, if there more than one service consumers on a device. Service consumer IDs are Internationalized Resource Identifiers (IRIs) that are unique on a given device. The complete utilized element to describe service consumer on DPWS devices is depicted in see figure 2.

There are several places in the device description metadata where this element can be embedded. It is possible to embed the element in the relationship section directly or a new metadata section can be defined. But with focus on DPWS implementations it is questionable if these can cope with additional elements inside the extension element in the relationship metadata. Thus for the implementation described in section 5, the short term solution is a new metadata section with the dialect `http://www.ws4d.org/services/UtilizableServices` that contains the host element of the relationship section and zero or more utilized elements.



Utilized	
1	Id
0..1	Device Types
0..1	Service Types

Figure 2. Utilized element

4.2. Utilized Services Manager

Through the Utilized Services Metadata a device can describe its service consumers. The next step is to control the service consumers. This is the task of the Utilized Services Manager (USM) which is a service that can be used to control the service consumers remotely. This service has basically two operations *GetServiceReferences* and *SetServiceReferences*. If an ID parameter of a service consumer is specified, the *GetServiceReferences* operation returns all references the corresponding service consumer is currently bound to. If the ID parameter is omitted, the operations returns all service consumers and their bound references. The *SetServiceReferences* requires an ID parameter to address the service consumer to control and a list of references to be applied to this service consumer. Different options are possible for reference manipulation to not always require to retrieve the current references of the consumer, modify it and overwrite the complete existing list. How the USM should deal with the references is indicated by an optional *Type* attribute in the *SetServiceReferences* message. If omitted the implied value is `http://www.ws4d.org/services/UtilizableServicesManager/merge`. This*merge* attribute value means that the passed references are merged with the already bound references. There are two other values *replace* and *exclude*. *Replace* replaces the current set of bound references with the passed references. *Exclude* excludes the references passed in the *SetServiceReferences* message from the currently set of bound references. Additionally the USM service offers an event to keep track of changes of the set of references a service consumer is bound to.

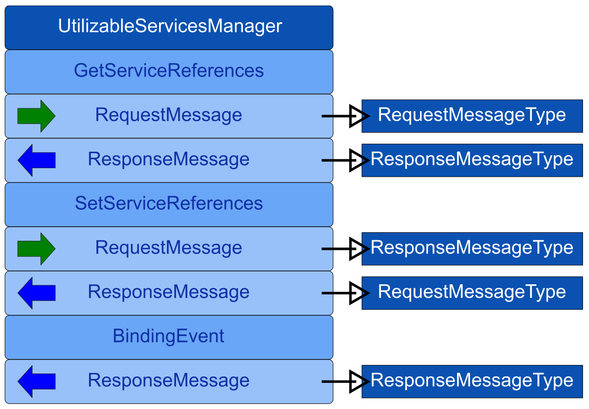


Figure 3. Utilized Services Manager Interface

The presented three operations of the USM form a interoperable way to control service consumers on DPWS devices. Thereby any Web Service capable framework or tool that can implement an USM client which features controlling the service consumers and thus build decentralized compositions of DPWS peers.

It is important to consider that the USM is not the only way to control bindings of consumers. Depending on the specific application, a consumer can retrieve service ref-

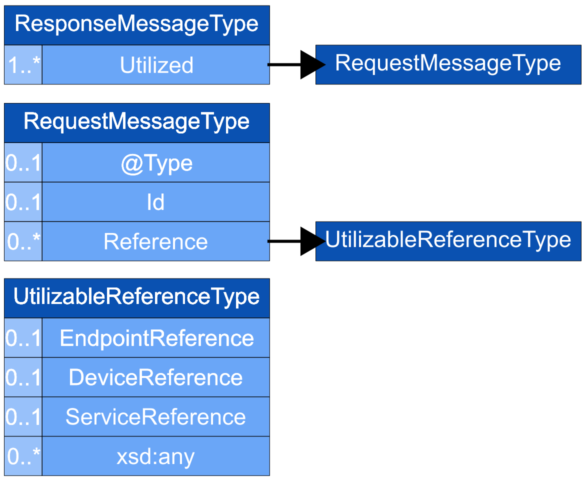


Figure 4. Utilized Services Manager Schema

erences by application specific configuration, DPWS device discovery or other means. The actual binding behavior of the consumers should be described within device type specifications. The USM enables third parties to get a global view on bindings in a DPWS peer composition but not on the actual data flow.

4.3. Service Composition

The approach described in this section provides not a complete component-oriented framework, but is a key enabling part to build architectures heading towards component orientation on top of DPWS. With the USM, a mechanism is available to control service consumers on devices. The next step is to build compositions on top of this mechanism in an abstract and flexible way in accordance to the *programming in the large* [8] paradigm. How to create such compositions is subject of ongoing research. An obvious solution is to use WS-BPEL and the BPEL4D-Extension described in [6]. But in general this topic needs further research efforts.

5. Implementation

In this section the implementation of the afore described concept in ws4d-gsoap [17] is described. This implementation bases on concepts and components that are explained in detail in the following subsections.

5.1. How to identify utilized services in DPWS peer applications

Handling the dynamic behaviour of DPWS in DPWS clients is a complicated issue. Especially when the applications scenario fully leverages the dynamic behaviour of discovery and description in DPWS. Depending on the applications scenario there are several possibilities how to implement an airconditioner client controlling an airconditioner device. The first applications scenario is quite simple and consists of retrieving the current temperature of all airconditioner devices available in the local network. To implement this scenario a client has to probe for all devices implementing the airconditioner device type. The client has to wait for a specific time frame dependent on the exact scenario and has the a list of references of device that responded to the probe request. It can then get the current temperature of the airconditioner devices. In this scenario there is no dynamic to handle, so it is straight forward to implement such a client. The second scenario is implies more dynamic. The airconditioner device offers an event to get notifications about change of the current temperature. The scenario requires a client that keeps track of the temporal temperature profile of all airconditioner devices available in the local network. In this case a client has to listen for discovery hello and bye messages and may probe for airconditioner devices in periodical intervals. This causes a higher implementation effort. To minimize this effort ws4d-gsoap includes trackers.

The concept behind of trackers in ws4d-gsoap is similar to service trackers in OSGi [14]. OSGi offers these service trackers to keep track of services offered by OSGi bundles. As OSGi bundles can be installed and uninstalled at runtime services can come and go at any time. So applications using such services have to cope with the dynamic of the availability of services. OSGi service trackers offer callbacks whenever a service of a specific type or another criteria changes availability. So applications can use the service tracker to react on the availability of services. In a similar way trackers in ws4d-gsoap do work. There are two types of trackers in implemented at the moment: devicetracker and servicetracker.

A device tracker tracks devices available in the network. For initialization a device tracker has an id and a device types parameter. The id is used to identify the tracker, and thus identify a specific service consumer. The optional type parameter can be used to filter the devices a tracker tracks by this device type. The device tracker internally manages a list of device references and can call callback whenever a tracked device changes its availability. This mechanism eases the implementation of dynamic client scenarions as the second example mentioned above. An as a useful extra feature it can be used to identify one type of consumers in dpws peer applications.

A service tracker tracks available services on devices available on the network. In contrast to device tracking in service tracking there is one more type involved: the service type. Thus the service tracker has an additional parameter to specify the filter for tracked services. Otherwise the service tracker has the same functionality as the device tracker.

In summary the tracker concept in ws4d-gsoap is used to identify consumers in DPWS peer applications.

5.2. Embedding Utilizable Services in device metadata

As described in subsection 4.1 the metadata about consumers can be embedded in several ways. The proper way would be to embedd it in the Relationship metadata section. As ws4d-gsoap does not offer this extension point at the moment an easier to implement approach was choosen. The consumer metadata is expressed in an addational metadata section with the dialect `http://www.ws4d.org/services/UtilizableServices`. A client can retrieve this metadata section as part of the device metadata.

In ws4d-gsoap all trackers are registerd at a tracker registry. This registry is used to identify all consumers that are announced in the device metadata. With every change in the device metadata the metadata numer is incremented and a discovery hello message with the new metadata numer is sent to the network. As the utilizable services metadata is part of the device metadata, trackers should be initialized in the setup stage of a device to avoid unintentional metadata changes.

5.3. Control service trackers with the utilizable services manager

The USM is the service to control the service bindings of an consumer that is an device or service tracker in the case of ws4d-gsoap. Trackers normally get their input from the discovery module of ws4d-gsoap. To plug in other mechanisms for configuration, etc. trackers can also be manipulated from other modules (references can be added and removed). This mechanism is used by the USM to add, remove and enumerate references managed a tracker. The USM uses the tracker registry to identify all registered trackers.

6. Discussion

As already mentioned in section 4 this approach is heading towards component orientation, but does not fully cover all issues that are solved in typical component-oriented frameworks. The component orientation is used to enable processes in service-oriented device applications that are not executed by centralized process engines but decentralized components that can directly communicate and only require configuration to build a process.

On the one hand, WS-BPEL, its DPWS extension called BPEL4D [6] and WS4D-PipesBox implement this centralized process engine approach where a centralized enginge executes a process description. In some device centric applications this centralized approach leads to implementations implications like centralized communication, bottlenecks, etc. that are not desired. On the other hand component-oriented frameworks offer the mechanisms to implement distributed processes but either bring the component orientation to a existing framework like iPOJO for OSGi or define their own framework where all components must be integrated like SCA. With the exception of iPOJO service-oriented process concepts and component-oriented frameworks are often not designed for device centric applications. So there is a need for a mechanism that better fits into the application domain of DPWS.

The dynamic and distributed nature of DPWS raises several issues when extending it towards component orientation. One example is dependency checking that could easily lead to dead or live locks in a distributed system as DPWS. These issues are not solved by this approach but must be considered in the application scenarios and higher level application protocols. So it is assumed that there are no dependencies between services hosted on a device and services consumed by a device. Thus the scope of this approach is to define an interoperable way to describe the services a DPWS peer can consume and a mechanism that third parties can control and configure which services a DPWS peer should use.

On top of this approach further mechanism can build a fully component-oriented framework or implement generic adapterd to integrate this approach into existing component-oriented frameworks.

7. Conclusion

This paper introduces a new approach to create applications, based on services provided by devices deployed with DPWS, in an abstract and dynamic way. The approach enables the creation of flexible service compositions that are not based on typical centralized process execution concepts but on distributed components that can directly communicat but can be configured remotely. This aproach is implemented with the ws4d-gsoap DPWS toolkit and compared to existing service-oriented process concepts and component-oriented frameworks. The resulting approach define an interoperable and reusable way to describe the services a DPWS peer can consume and a mechanism how third parties can control and configure which services a DPWS peer should use.

Acknowledgment

This work has been funded by German Federal Ministry of Education and Research (BMBF) under reference number 01IS08003 in the context of the European ITEA2 OASMI project.

References

[1] Web Services on Devices, 2008. `http://msdn.microsoft.com/en-us/library/aa826001(VS.85).aspx`.

[2] Service-oriented Architectures for Devices, 2009. `http://www.soa4d.org`.

[3] Web Services for Devices, 2009. `http://www.ws4d.org`.

[4] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web services business process execution language version 2.0, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[5] A. Bobek. *Serviceorientierte Infrastruktur für vernetzte Dienste und eingebettete Geräte*. PhD thesis, Fakultät für Informatik und Elektrotechnik der Universität Rostock, Germany, Dezember 2008.

[6] H. Bohn. *Web Service Composition for Embedded Systems - A WS-BPEL Extension for DPWS*. PhD thesis, Fakultät für Informatik und Elektrotechnik der Universität Rostock, Germany, Februar 2009. ISBN: 978-3-86844-108-6.

[7] H. Bohn, A. Bobek, and F. Golatowski. SIRENA - Service Infrastructure for Real-time Embedded Networked Devices: A service oriented framework for different domains. In *International Conference on Networking (ICN)*, 2006.

[8] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM.

[9] W. Dostal, M. Jeckle, I. Melzer, and B. Zengler. *Serviceorientierte Architekturen mit Web Services*. Elsevier, 2005.

[10] C. Escoffier and R. S. Hall. Dynamically adaptable applications with ipojo service components. In *Software Composition*, pages 113–128, 2007.

[11] F. Jammes, A. Mensch, and H. Smit. Service-oriented device communications using the devices profile for web services. In *3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC05) at the 6th International Middleware Conference*, 2005.

[12] A. Malhotra, K. Warr, D. Davis, and W. Chou. Web services eventing (WS-eventing). W3C working draft, W3C, Dec. 2009. http://www.w3.org/TR/2009/WD-ws-eventing-20091217.

[13] Microsoft, Intel, Ricoh, Lexmark. *Devices Profile for Web Services*, 2006. `http://schemas.xmlsoap.org/ws/2006/02/devprof/`.

[14] OSGi Alliance. *OSGi Alliance: OSGi Service Platform, Core Specification, Release 4, Version 4.2*. OSGi Alliance, 2009.

[15] World Wide Web Consortium (W3C). *Web Services Architecture*, 2004.

[16] E. Zeeb, A. Bobek, H. Bohn, and F. Golatowski. Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services. In *2nd International IEEE Workshop on SOCNE 07*, 2007.

[17] E. Zeeb, A. Bobek, H. Bohn, S. Prüter, A. Pohl, H. Krumm, I. Lück, F. Golatowski, and D. Timmermann. Ws4d: Soa-toolkits making embedded systems ready for web services. In *Open Source Software and Productlines 2007 (OSSPL07)*.