Journal of Universal Computer Science, vol. 20, no. 8 (2014), 1109-1129
submitted: 4/4/14, accepted: 23/7/14, appeared: 1/8/14 © *J.UCS*

Internet of Things Aware WS-BPEL Business Processes - Context Variables and Expected Exceptions

Dulce Domingos, Francisco Martins, Carlos Cândido
(Large-Scale Informatics Systems Laboratory (LASIGE), Faculty of Sciences
University of Lisbon, Portugal
dulce,fmartins@di.fc.ul.pt, c.candido85@gmail.com)

Ricardo Martinho
(School of Technology and Management, Polytechnic Institute of Leiria
Portugal
ricardo.martinho@ipleiria.pt)

Abstract: Business processes can use Internet of Things (IoT) information to monitor context data in real-time and to respond to changes in their values in a timely fashion. For this matter, business process definition and execution languages should foresee an easy way for process modelers to define which values to monitor, and which automatic behaviors to adopt when these values change. In this paper, we propose the use of context variables to monitor sensor values, as well as a *when-then* language construct to detect and handle changes in these values within business processes. We define a Web Services Business Process Execution Language (WS-BPEL) extension to convey these constructs, and implement then using a "BPEL language transformation" approach. With these contributions, process modelers can define IoT-aware business processes avoiding the increase of process complexity and keeping their focus on modeling the processes' main logic. In addition, the language transformation approach assures the portability of processes using our constructs amongst WS-BPEL execution engines.
Key Words: IoT, business process, language constructs, context variable, WS-BPEL extension
Category: C.3, H.4.1

1 Introduction

The Internet of Things (IoT) provides information about what is actually happening in the real world, in real time. Business processes can gain a competitive advantage by using this information during their execution. For instance, in [Jedermann and Lang, 2008], the authors present a case study that uses temperature information to determine the final destination of strawberry pallets, considering their remaining shelf life.

Following a more reactive paradigm, business processes can monitor real world information and get notified about context (e.g., environment) changes [Predic and Stojanovic, 2012]. With context information, business processes can even automatically change their execution to react to new conditions, as soon as they happen [Zhang et al., 2012]. In the strawberry example mentioned above, it would be possible to define a logistics business process to automatically change the delivery route or the destination if the strawberry temperature rises above a threshold during delivery.

To facilitate the access to IoT information and functionalities, recent works expose them as web services that can be implemented directly in the sensor devices or in a middleware layer [Zeng et al., 2011]. The service-oriented approach has the advantage of enhancing interoperability and of encapsulating heterogeneity and specificities of sensor devices, such as communication protocols.

The *web services business process execution language* [OASIS, 2007] (WS-BPEL) is the *de facto* standard language for defining processes through orchestration of web services. Using web services, sensor information can be easily integrated into processes via a synchronous request/reply paradigm.

However, if processes need updated information about environment changes, modelers have to include ad hoc operations to the process definition to handle such requirements. For instance, to obtain sensor information periodically, modelers need to explicitly program this (often cumbersome) behavior, deviating their modeling focus from the main process logic. Moreover, WS-BPEL lacks a native mechanism to monitor variable changes, so it makes it difficult to define processes that react to environment changes. Current approaches extend WS-BPEL with context variables that are automatically aware of changes to sensor-reported values. They also accommodate new language constructs to handle expected exceptions. However, these approaches implement WS-BPEL extensions by changing the WS-BPEL engine behavior, preventing process portability.

The work we present in this paper simplify the use of updated sensor context information in business processes and enhance process reactivity. We define a WS-BPEL extension with context variables that can be updated automatically. We achieve this either through the synchronous request/reply paradigm using the WS-ResourceProperties standard [OASIS, 2006b], or via the asynchronous publish/subscribe paradigm using the WS-Notifications standard [OASIS, 2006a]. Our extension also proposes a *when-then* new construct to handle expected exceptions. We realize this extension following a language transformation approach, fully compliant with any WS-BPEL engine.

The paper is organized as follows: the next section presents an overview of WS-BPEL and Section 3 describes a motivating scenario. Section 4 discusses related work. In Sections 5 and 6 we define our WS-BPEL extension and its implementation, respectively. To evaluate our approach we developed a prototype and use business process metrics as described in Section 7. Finally, Section 8 concludes the paper and presents future work.

2 Overview of WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL) is the OASIS standard executable language for defining business processes through web service orchestrations [OASIS, 2007]. A business process definition includes two elements: a WSDL file that describes the business process functionalities (web services) with their message data structures, service addresses, among others, and a WS-BPEL file that defines the business process logic.

WS-BPEL includes different types of activities, such as flow control activities (*If, While, Pick, Flow*), communication activities (*Receive, Reply, Invoke*), assign value activities (*Assign*), fault handlers (*Throw, Rethrow*), to name a few. We can declare process variables of any primitive or complex types (a composition of primitive or complex types), and of message types. Message variables are used almost exclusively in communication activities. Variables can be global or local, if declared within a *Scope*.

Processes in WS-BPEL export and import functionalities by using web services. Web services are modeled as *partnerLinks*, characterized by a *partnerLinkType*, which is defined in the WSDL definition. A *partnerLinkType* specifies the role and the type of a partner. An input communication activity is associated with the process's *MyRole* and an output communication activity is associated with the partner's *PartnerRole*.

In order to distinguish process instances, WS-BPEL provides the *correlation* mechanism. A *correlationSet* is defined by (1) the rule set (one per message type) that determines the message fields used to identify an instance; and (2) the primitive data type that will be used. The *correlationSet* is associated with communication activities. Each *correlationSet* can be initialized once and, if we use it in an *Invoke* activity, we have to define when the *Correlation* is established: on the sending operation, upon response, or on both. The *CorrelationSets* property defines, through XPATH, the message elements that identify uniquely each conversation (i.e., each process instance).

The WS-BPEL standard foresees native extension mechanisms by allowing namespace qualified attributes to appear in any WS-BPEL element, and by allowing elements from other namespaces to appear within WS-BPEL defined elements. In addition, WS-BPEL provides two explicit extension constructs: *extensionAssignOperation* and *extensionActivity*. All extensions used in a process must be declared; this is made by inserting the namespaces associated with the extensions into the *Extensions* construct language, along with the *MustUnderstand* attribute (values *yes* or *no*), which states whether the process execution engine has to support the extension.

There are two different options to realize an extension [Kopp et al., 2011] within WS-BPEL. The first one addresses a "WS-BPEL Language Transformation", where extension constructs are translated into standard WS-BPEL

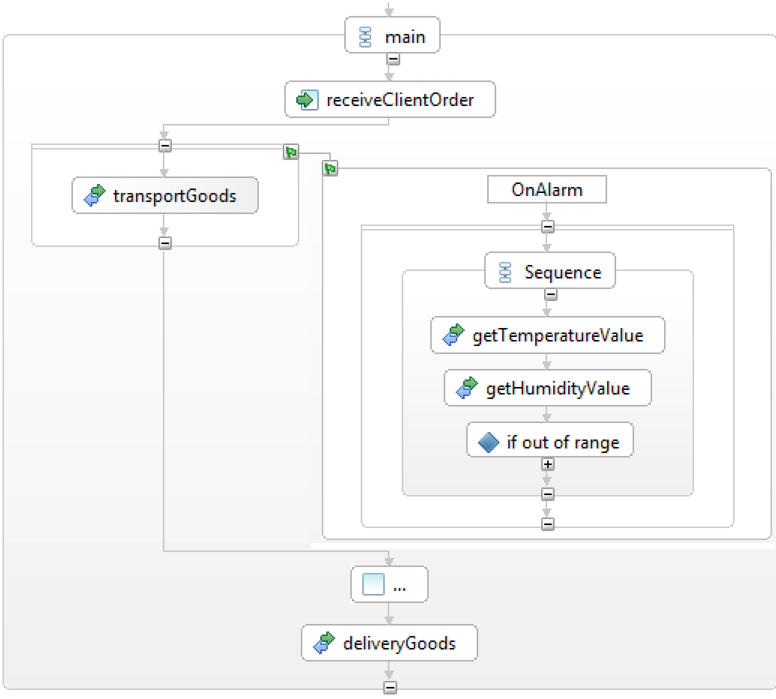

Figure 1: Use case scenario with repeated sensor values checking

constructs. The generated standard WS-BPEL code can be deployed on a process execution engine that ignores the extension. This option is only feasible if new constructs are expressible with a set of standard constructs. The second extension option goes for the "WS-BPEL runtime engine", where the extension is realized by changing the process execution engine in order to support the additional functionalities.

3 Use case scenario

As a motivating scenario, we present a typical use case of perishable goods transportation, such as strawberries. A distribution company receives client orders and performs the transportation of goods. During transportation, the company monitors the temperature and the humidity of the goods with sensors. If these values change in such a way that it represents a threat to the good's quality, the company can, for instance, change the route to a faster one or change the delivery destination to a closer client. To achieve this behavior using WS-BPEL, process instances have to repeatedly get and check sensor values, as sketched in Figure 1. In this example, process instances interact with both sensors using the synchronous request/reply paradigm.

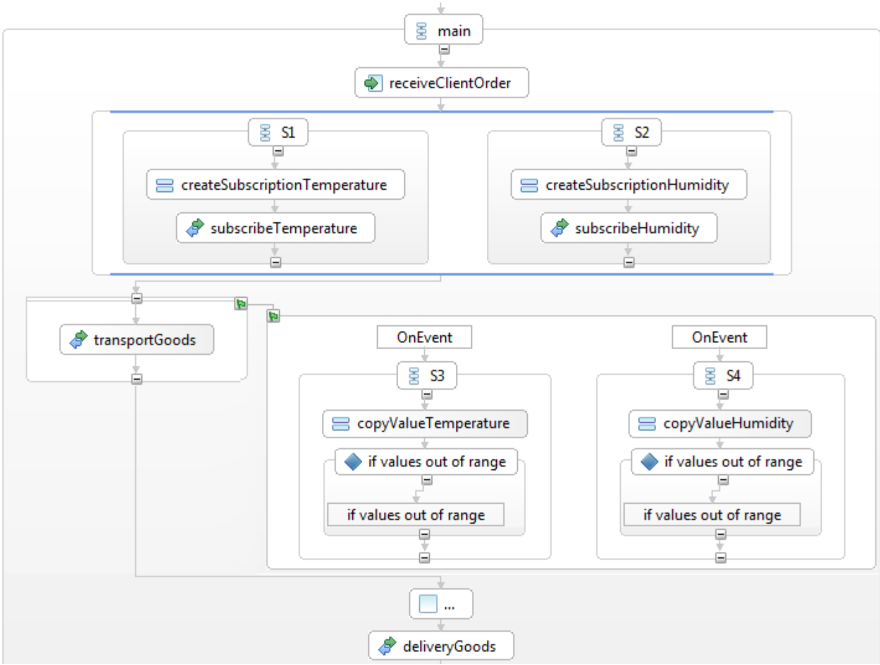

Figure 2: Use case scenario with publish/subscribe sensor interaction

However, this interaction can be improved in case sensors make available an asynchronous publish/subscribe interface that notifies process instances only when a value change occurs. To receive sensor notifications, processes have to previously subscribe them. If, in our use case scenario, we replace the request/reply paradigm by the publish/subscribe paradigm, when process instances receive, for example, a notification from the temperature sensor, they need to keep track of previous humidity values in order to evaluate expressions that depend on both readings (*e.g.*, an *if* guard). In summary, the modeler needs to include additional operations to subscribe temperature and humidity values, to receive sensors notifications, and to keep an history of their previous values, as illustrated in Figure 2. Indeed, this additional behavior scatters process definitions with new operations, increases process size and complexity, and diverts the attention of the process engineer from the main business process logic.

4 Related Work

Current IoT technology exposes physical objects information and functionalities as web services [Zeng et al., 2011]. Web services encapsulate heterogeneity and specificities of physical objects and are well suited to the modeling languages that define business processes as service orchestrations, such as Business Process Model and Notation (BPMN) [OMG, 2011] and WS-BPEL.

Traditionally, context information is obtained according to a synchronous request/response paradigm, and business processes use it in predefined points. They use context information to: (1) determine the services that compose processes [Yu and Su, 2009]; (2) choose between multiple implementations for a specific service [Ranganathan and McFaddin, 2004]; or (3) determine whether a service should participate in future compositions [Karastoyanova et al., 2005].

Following a domain-specific language approach, some authors propose the explicit integration of IoT concepts into business process models. In [George, 2008, George and Ward, 2008, Domingos et al., 2013], the authors extend WS-BPEL with context variables. George and Ward define context as an environment state, external to the process, whose value can change independently of the process lifecycle, and can influence process execution. In [Meyer et al., 2013, Meyer et al., 2012], the authors extend BPMN with seven new modeling concepts (*IoT Activity, Sensing Activity, Process Resources, Physical Entity, Real World Data Object/Store, Mobility Aspect, and IoT Process Ratios*). In particular, a *Real World Data Object* represents a temporarily stored data object of a running process instance, which was generated by an *IoT Device*, while a *Real World Data Store* represents persistent data.

In addition, we can find in the literature some works that automatically synchronize context variable values with sensor information. They differ in the interaction paradigm they use to communicate with sensors (synchronous request/reply or asynchronous publish/subscribe) and in the way they realize the WS-BPEL extension: by using a “WS-BPEL Language Transformation” or a “WS-BPEL runtime extension” [Kopp et al., 2011].

In [George, 2008, George and Ward, 2008], the authors propose the first IoT domain specific WS-BPEL extension with context variables. They use WS-BPEL language extension mechanisms by adding extension attributes to the standard variable construct. These variables are updated using the publish/subscribe paradigm following the WS-Notification standard. The authors realize their extension by changing the ActiveBPEL 4.1 engine [Informatica, 2014]. When a process uses a context variable with an *invoke* activity, the engine performs the *subscribe* operation and becomes responsible for updating the variable when it receives notifications. They also use the Apache Muse tool [Apache Muse, 2014] to handle subscription and notification operations and to distinguish process instances. Despite this approach reduces the modeling effort, process definitions must explicitly contain the *invoke* operation with the variable to trigger the subscription operation. In addition, the portability of the extension is limited both because it is a runtime extension and it depends on the Apache Muse tool.

To explicitly model context influence on workflows, in [Wieland et al., 2007], the authors propose Context4BPEL, a domain generic WS-BPEL extension defined according to the WS-BPEL extension mechanisms. This extension includes features to: (1) manage context events to allow the asynchronous reception of events; (2) query context data; and (3) evaluate transition conditions based on context data. Context4BPEL is also implemented as a runtime extension and consequently needs adaptations to the process execution engine. Moreover, the context information management depends on the Nexus platform.

In [Wieland et al., 2009], the authors propose a WS-BPEL extension that includes reference variables, a concept similar to context variables. With this kind of variables, services can exchange pointers to variables instead of their values. Pointers are represented with EndPoint References (EPR). According to the value of an attribute of the extension, references are evaluated (1) upon activation of a WS-BPEL's *scope* element; (2) before variables are used; (3) periodically; or (4) through an event sent from an external service. This extension is realized as a language transformation approach, replacing references with WS-BPEL variables, inserting links to partners and interaction activities. Reference evaluation depends on the Reference Resolution Service (RRS), a specific service of the platform the authors propose.

Krizevnik and Juric [Krizevnik and Juric, 2012] extend WS-BPEL with data-bound variables, which are automatically synchronized. They use Data Access Services as data providers and implement their prototype as an extension of the ActiveBPEL engine.

In [Domingos et al., 2013], we present our first approach to an IoT-aware WS-BPEL extension with context variables. We realize it using a language transformation approach and a publish/subscribe paradigm to interact with sensors.

Mateo, Valero, Díaz [Mateo et al., 2012] formalize a fragment of WS-BPEL together with Web Services Resource Framework (WSRF) to incorporate distributed resources into WS-BPEL. They provide an operational semantics (by means of a label transition system) for their proposed language, which constitutes an alternative approach to extend WS-BPEL in a formal way.

Business processes also need to handle exceptions to react to new conditions, as soon as they happen. Unexpected exceptions require user intervention in order to change a process instance, or even to change the process definition and, in some situations, its running instances. Reichert and Rinderle address the problem of how these changes can be realized in a correct and consistent manner in WS-BPEL [Reichert and Rinderle, 2006].

Expected exceptions refer to predictable deviations from normal behavior of the process. These deviations can be addressed directly by adding alternative flow paths. WS-BPEL has some constructs with this purpose: *Throw*, *Catch*, and *Rethrow*. As in the Java language, within its normal flow, processes can launch exceptions using the *Throw* constructor. With the *Catch* construct, modelers define alternative flows to handle them. As the *Throw* construct is synchronous, if the modeler wants to monitor the value of a variable, she has to add process operations to periodically check it.

In the workflow domain, Event-Condition-Action (ECA) statements are used to define the conditions to be monitored and the activities to be performed to handle them [Casati et al., 1999]. For WS-BPEL, these statements are provided by following two distinct approaches. The first one includes additional constructs to the language. George and Ward add the *conditionWithTimeout* construct to the WS-BPEL [George and Ward, 2008], while Domingos et al. propose the *when* construct [Domingos et al., 2013]. George and Ward realize a runtime extension, while Domingos et al. propose their first step to realize a WS-BPEL language transformation, still using however, the *listeners* of the Apache ODE [Apache ODE, 2014] to detect variable values modifications. The other approach specifies exceptional behavior with rule-based languages. Liu et al. support ECA rules in WS-BPEL [Liu et al., 2007]. Zeng et al. propose a similar approach to separate normal behavior from exceptional behavior [Zeng et al., 2005].

In this work we define a WS-BPEL extension with context variables that can be automatically updated by using the synchronous request/reply paradigm or the asynchronous publish/subscribe paradigm. This extension also includes the *when-then* construct to handle expected exceptions. Unlike previous approaches, we realize this extension following a WS-BPEL language transformation process. This way, this extension is fully compliant with any WS-BPEL engine.

5 IoT WS-BPEL extension definition

Business processes can gain competitive advantage by using updated information about real-world context that, for instance, can be provided by sensors. However, with current business process modeling languages, to get, maintain, and monitor this information, process modelers have to include several additional activities to interact with sensors and to monitor their values.

The extension we propose includes two additional language constructs to facilitate the use of context information within business processes: (1) *context variables* to capture and to maintain sensor values automatically; and (2) a *when-then* construct that monitors sensor data changes and specifies how processes react to exceptional conditions.

Our extension builds on top of standard WS-BPEL extension mechanisms. Its alias is *iotx* and we declare the *mustUnderstand* element set to *no*, as execution engines do not need to understand this extension, since it is a language transformation extension as illustrated in Listing 1 (see details in Section 6).

```
  xmlns:iotx="http://bpel.iot.extensions"
  <bpel:extensions>
    <bpel:extension namespace="http://bpel.iot.extensions" mustUnderstand="no"/>
  </bpel:extensions>
```

Listing 1: Declaration of the extension and its namespace

```
<variable name="BPELVariableName"
  iotx:comunicationType="request/response"
  iotx:sourceEPR="URL"
  iotx:resourceProperty="ResourceName"
  iotx:refreshTime="Time"
</variable>
```

Listing 2: Context Variable extension syntax for request/reply interaction

5.1 Context variable definition

Context variables simplify the access to context information within WS-BPEL processes. We use the existent WS-BPEL *variable* construct to keep sensor values, and extend it with new attributes to automatically update their values. This way, process modelers use the same language construct when declaring context variables, and only have to complement the information regarding the original *variable* construct.

We support two interaction mechanisms to automatically update variable's values. The first one updates the variable periodically using a synchronous request/reply interaction and following the *WS-ResourceProperties* standard. The second uses an asynchronous publish/subscribe interaction according to the *WS-Notifications* standard. Attribute *communicationType* stores this interaction type used to update the variable, namely request/response or publish/subscribe. For the sake of simplicity, we present additional attributes for each type of interaction in two different Listings (Listing 2 and Listing 3).

The synchronous request/reply type of interaction needs additional attributes to establish the communication according to the *WS-ResourceProperties* standard: the web service that provides the sensor information, the resource property element, and the *refreshTime* that defines the update frequency of the variable.

For the asynchronous publish/subscribe interaction, the new attributes represent the information needed to establish the subscribe operation according to the WS-Notifications standard: the web service that provides the subscribe operation and the subscription topic.

Listing 4 illustrates the definition of the two context variables of our use case scenario. The *humidityVar* variable is updated synchronously each five minutes, while the *temperatureVar* variable is updated when the temperature changes are triggered by sensor notifications.

```
<variable name="BPELVariableName"
cc iotx:comunicationType="publish/subscribe"
  iotx:publisherEPR="URL"
  iotx:topic="Topic" ?
</variable>
```

Listing 3: Context Variable extension syntax for publish/subscribe interaction

```
<bpel:variables>
  <bpel:variable name="humidityVar" type="xsd:int"
    iotx:comunicationType="request/response"
    iotx:sourceEPR="http://192.168.1.52:8081/axis2/services/SensorService"
    iotx:ResourceProperty="Humidity"
    iotx:refreshTime="PT5M0S"/>

  <bpel:variable name="temperatureVar" type="xsd:int"
    iotx:comunicationType="publish/subscribe"
    iotx:publisherEPR="http://192.168.1.52:8081/axis2/services/SensorService"
    iotx:topic="Temperature"/>
</bpel:variables>
```

Listing 4: Definition of context variables

5.2 The when-then language construct

We define the *when-then* language construct as a new WS-BPEL section, similar to fault and event handlers. This construct implements a guarded activity. It contains a condition and an activity, as shown is Listing 5. The activity is executed when the condition becomes true.

Listing 6 exemplifies the definition of an expected exception with the *when-then* construct. We use the *temperatureVar* context variable to define the *when-then* condition. When its value is above 35, the *sequence* activity is executed.

The next section describes the realization of this extension using a language transformation approach.

6 IoT WS-BPEL extension realisation

WS-BPEL extensions can be realized by following a language transformation approach or by changing the runtime engine, as previously mentioned. As we are able to represent the new language constructs using existing WS-BPEL standard constructs, we follow the first approach. It has the advantage of being independent of the runtime engine, keeping processes more reusable and portable

```
<iotx: when standard−attributes>
  <bpel:condition expressionLanguage="anyURI"?>
    bool−expr
  </bpel:condition>
    activity
</iotx:when>
```

Listing 5: When-then language construct syntax

```
<iotx:when name="temperatureCondition">
  <bpel:condition>$temperatureVar > 35</bpel:condition>
  <sequence name="test">
    <empty name="empty1"> </empty>
    <empty name="empty2"> </empty>
  </sequence>
</iotx:when>
```

Listing 6: Example of the when-then construct

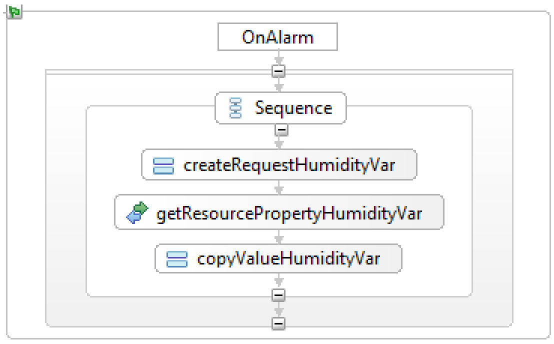

Figure 3: Language transformation - request/reply interaction

between different WS-BPEL engines. The transformation includes changing the WS-BPEL process definition. As it adds the invocation of external services to obtain sensor data, we also have to create WSDL files to describe these services.

6.1 Context variables

Within the WS-BPEL file, our transformation replaces context variables by standard variables, removing the extension attributes. Furthermore, we add the *partnerLinks* we use to interact with sensors and the imports of the WSDL files.

In the following we detail the transformations we realize for context variables considering each type of interaction.

6.1.1 Context variables with request/reply interaction

For context variables that use the request/reply interaction, our language transformation adds, to the WS-BPEL process definition, the operations we need to periodically request the sensor value and update the context variable. These operations need to run in parallel with the activities of the original process definition. This way, we add them inside an *eventHandlers* section with an *onAlarm*, which repeats according to the *refreshTime* attribute, as illustrated in Figure 3.

The sequence we define within the *onAlarm* includes an *Assign* activity to initialize the message we use to send to the sensor. This message uses the information of the *ResourceProperty* attribute of the context variable.

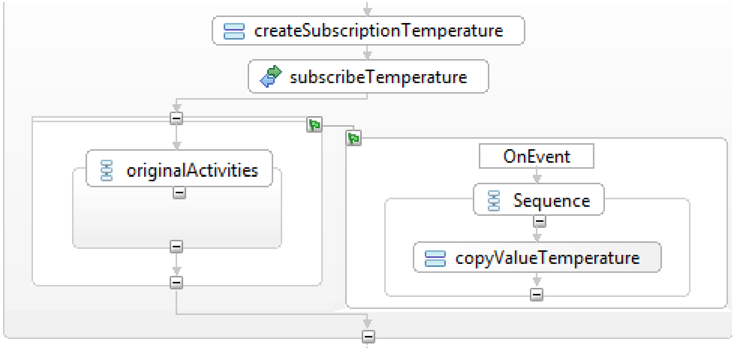

Figure 4: Language transformation - publish/subscribe interaction

The *invoke* activity is used to get the sensor value by invoking the *GetResourceProperty* operation according to the *WS-ResourceProperties* standard. This *invoke* activity uses two messages. The previous *assign* activity initializes the first message, which is used as the request message. The other message is used to keep the response.

Finally, the *copyValueHumidityVar* assign activity gets the value from the *getResourceResponseHumidityVar* message and updates the variable value. Taking into account that the service we use follows the *WS-ResourceProperties* standard, we only need to import its WSDL file.

6.1.2 Context variables with publish/subscribe interaction

For context variables that use the publish/subscribe interaction, our language transformation adds, to the WS-BPEL process definition, the operations we need to perform the subscribe operations and to receive notifications in order to update the process variable value.

We invoke the subscribe operation before the first activity of the process/scope and we perform the reception of notifications in parallel with the activities of the original process definition. We add the activities to handle the reception of notifications inside an *eventHandlers* section with an *onEvent* (see Figure 4).

The subscribe operation is done with an *Invoke* activity. This activity calls the *publisherEPR* defined in the context variable. As the subscribe operation is a two-way operation, we define two variables: the *inputVariable* and the *outputVariable*. Before the *Invoke* activity, we use an *Assign* activity to initialize the message that the *Invoke* activity sends to the publisher. We format this message according to the WS-Notification standard. The message is initialized with the topic declared in the context variable and the EndPoint Reference (EPR) to where the publisher sends notifications (the NotificationConsumer). The EPR is generated by concatenating the process name with the name of the context

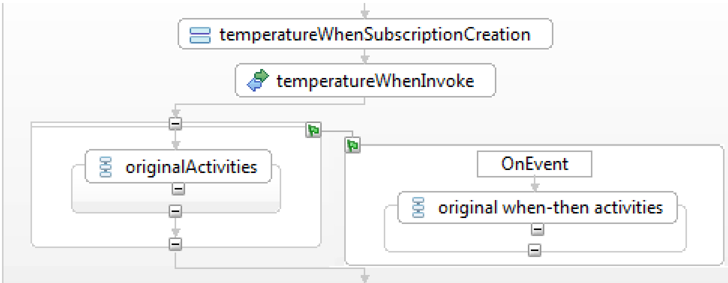

Figure 5: Language transformation for the *when-then* construct

variable. The *output* variable is initialized in the *Invoke* response. As we start the *Correlation* between calls to web services and the process instance within the *Invoke* response, we declare its initialization here.

Processes receive notifications through an *onEvent* event handler. This event handler uses a variable to store the notifications and the *Correlation*.

When the notification message arrives, the *scope* activity specified in the corresponding event handler is executed. It includes an *Assign* activity to copy the value of the notification message to the context variable.

Each context variable is related with two services: the subscribe service and the service to where notifications are sent (the consumer service). We define these services in an additional WSDL file, which the transformed WS-BPEL process imports. This way, we avoid modifying the original WSDL files from these services. We get the address of the subscribe service directly from the definition of the context variable, and we generate the address of the consumer service by concatenating the process name with the name of the context variable.

This WSDL file also includes the *Correlation* properties we use in the WS-BPEL transformation. Considering that all process instances use the same port to receive notifications, we use the *Correlation* properties to distinguish them.

We define a *correlationSet* for each context variable, since each one maps to a different subscription. We use the *Correlation* with two messages (*Subscribe Response* and *Notify*). Thus we define two rules and we use them in all the *correlationSets*. The rules state that, for each message, the correlations use the field *ReferenceParameters* of the element *SubscriptionReference*. The data type has to be the same as the field *ReferenceParameters*, i.e., *anyURI* (any type).

6.2 When-then construct

We also implement the *when-then* construct with a language transformation approach. However, we use an auxiliary web service to evaluate the *when-then* conditions, when changes to a variable value occurs.

In the following we detail our *when-then* construct realisation, which includes replacing the *when-then* by a set of standard constructs, detecting variable value modifications, creating the WSDL file, and calling the auxiliary web service.

When editing the WS-BPEL process file, we remove the *when-then* construct and add the definitions we need: operations for *when-then* construct; two message variables; *partnerLink*; *CorrelationSet*; and the WSDL file imports.

To execute the *when-then* process logic in parallel with the main process logic, we use an *onEvent* event handler. As we can see in Figure 5, before that, we add the *Assign* operation to initialize the message we use to invoke the register operation in the auxiliary web service. This message has the *when-then* condition, the EPR used in the *onEvent* to receive the message notifying that the *when-then* condition becomes true, and the identification of the process instance. The identification of the process instance depends on a variable provided by the WS-BPEL engine (Apache ODE, in our case), *$ode:pid*. However, other process engines also provide this type of variable.

After the *Assign* operation, we add an *Invoke* activity to call the *RegisterWhen* operation of the auxiliary web service. This activity also uses a *CorrelationSet*, which distinguishes processes instances through their identification (InstanceId).

The *onEvent* receives the messages sent by the auxiliary web service, when the *when-then* condition becomes true. It uses a *CorrelationSet* with the process instance identification. When the *onEvent* receives a message, it executes the original *when-then* activities, which define the *scope* activity of the event handler.

To detect variable value modifications, we follow two different approaches. In the first one, we use the *event listeners* of the Apache ODE WS-BPEL engine. ODE generates events, such as *VariableModificationEvent* events, that can be used to monitor what is happening in the engine, and supports the registration of *event listeners* to analyze produced events and to do whatever operations we want. The event listener we developed informs the auxiliary web service about *VariableModificationEvent* events. Against this information, the auxiliary web service evaluates registered *when-then* conditions.

As this approach depends on the *event listeners* of Apache ODE, we realize a second and more process engine-independent approach. Following a language transformation, we add, to the WS-BPEL process definition, a set of two activities (*assign* and *invoke*), to each activity that can modify a variable value. The *invoke* activity informs the auxiliary web service about a variable value modification. Before that, the *assign* activity creates the message the *invoke* activity uses. This message includes the variable name, its new value, and the process instance identification.

The WSDL file includes two web services interfaces: (1) the auxiliary web service, which has two operations: the operation to inform about modifications of a variable value; and the operation to register *when-then* conditions; and (2) the web service that corresponds to the *onEvent* event handler used to inform process instances when a *when-then* condition becomes true. The EPR of this web service is generated by concatenating the process name with the name of the *when-then* construct. Consequently, the *when* name attribute is mandatory. The WSDL file also includes the messages operations the *partnerLinkType*, and the correlation rules. Correlation rules only use the process instance identification.

The auxiliary web service evaluates the value of *when-then* conditions. It provides the *RegisterWhen* operation—to register conditions clients want this service to evaluate—and the *UpdateVar* operation, which clients use to inform about a variable value modification. It calls the *UnlockWhen* operation to notify the process instance when a *when-then* condition becomes true.

As stated before, to distinguish process instances we use the process instance identification the process engine provides.

7 Prototype and evaluation

This section presents the implementation of the prototype of our WS-BPEL extension. We also present the evaluation of our approach considering the size and complexity reduction that process definitions achieve when using our WS-BPEL extension, as well as the results of some performance tests.

7.1 Prototype

As referred above, we implement our WS-BPEL extension following a language transformation approach. We use the Eclipse IDE and its BPEL Designer plugin [Eclipse IDE, 2014, BPEL Designer Project, 2014] to define WS-BPEL extended processes. We perform the model transformation with Extensible Stylesheet Language Transformations (XSLT) [Clark et al., 2007] and the Saxon Home Edition 9.4 - XLST Processor [Saxon Home Edition, 2014]. XSLT is a specification that defines the syntax and semantics of a language to transform and render XML documents. It is designed for use as part of the Extensible Stylesheet Language (XSL), which is a style language for XML. XSL includes an XML vocabulary to specify formatting and uses XSLT to describe the document transformation. We choose this XLST processor since it supports XSLT 2.0, a XSLT version that can generate more than one output file, which is an advantage for our prototype since we have both WS-BPEL files and WSDL files.

Finally, the process execution engine we use is the Apache Orchestration Director Engine (Apache ODE) [Apache ODE, 2014], a free open-source engine from the Apache Foundation. We install it in the Apache Tomcat web

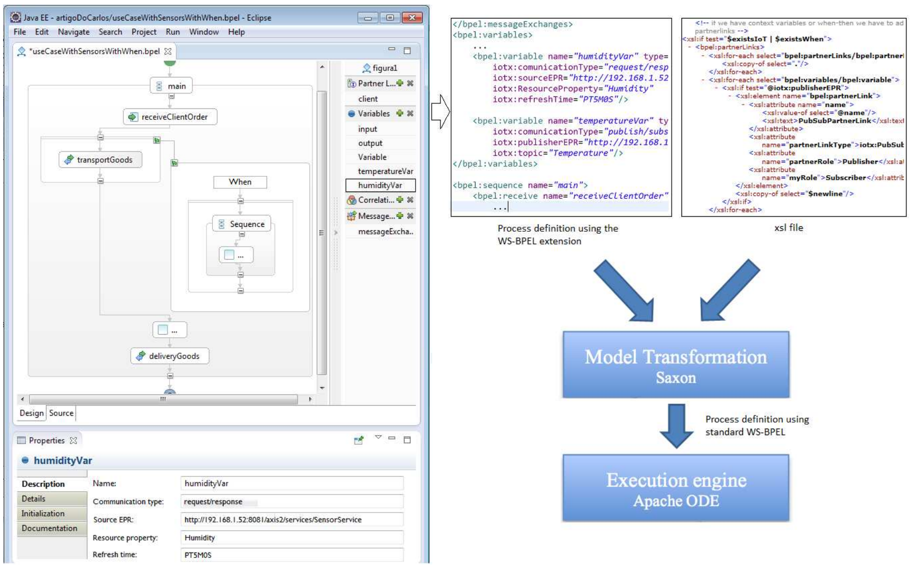

Figure 6: Toolchain prototype

server [Apache Tomcat, 2014]. In Figure 6 we illustrate our toolchain prototype and how it performs language transformations. These tools are available at http://gloss.di.fc.ul.pt/pati.

7.2 Complexity evaluation

Our WS-BPEL extension aims at facilitating process definitions by automatically synchronizing process variables with sensor values, without needing to explicitly define all interactions. In addition, we also provide the *when-then* construct to define expected exceptions, whose conditions can include context variables.

In this section we evaluate our proposal by using WS-BPEL process metrics. We can find in the literature several metrics to evaluate business process complexity [Muketha et al., 2010]. However, as far as we know, only two authors propose metrics specifically adapted to WS-BPEL processes. Cardoso [Cardoso, 2007] proposes the control flow complexity (CFC), an adaptation of the McCabe's cyclomatic complexity metric. While this metric assigns the same semantics to all decisions nodes, CFC distinguishes the various structured activities (*sequence*, *switch*, *while*, *flow* and *pick*). CFC metric has been validated with Weyuker' s properties and with several experiments [Muketha et al., 2010]. Mao adapts cognitive weights for WS-BPEL to measure cognitive complexity (CC) of WS-BPEL processes [Mao, 2010].

To evaluate our solution we also use two activity complexity metrics. These metrics only calculate the number of activities a process has. According to Car-

```
CFC(P1)=CFC(main sequence)=
        =CFC(receiveClientOrder)+CFC(transportGoods)+CFC(OnAlarm)+...=
        =1+CFC(transportGoods)+ 2^(n-1)xCFC(Sequence of OnAlarm)+...=
        =1+CFC(transportGoods)+1x(1+1+1)+..., where 2^(n-1) is the CFC of an event
             handler with n events.

CC(P1)=CC(receiveClientOrder)+CC(transportGoods)+CC(OnAlarm)+...=
       =1+CC(transportGoods)+3+CC(getTempValue)+CC(getHumidityValue)+CC(if)+...=
       =1+CFC(transportGoods)+3+1+1+2+..., where CC(OnAlarm)=3 and CC(if)=2.
```

Listing 7: P1 process metrics

```
CFC(P2)=CFC(main sequence)=
        =CFC(receiveClientOrder)+{CFC(flow)}+CFC(transportGoods)+CFC(eventHandlers
             )+...=
        =1+n!x[CFC(S1)+CFC(S2)]+CFC(transportGoods)+ 2^(m-1)x(CFC(S3)+CFC(S4))
             +...=
        =1+2!x(2+2)+CFC(transportGoods)+2x(1+1+CFC(then)+1+1+CFC(then))+...
                  where CFC(flow)=n!, CFC(eventHandlers) = 2^(m-1), n is the number of
                       flow
                  activities and m is the number of events.

CC(P2)=CC(receiveClientOrder)+CC(flow)+CC(transportGoods)+CC(eventHandlers)+...=
      =1+(4+1+1+1+1+4)+CC(tranportGoods)+(3+1+2+CC(then)+1+2+CC(then))
           +...
                where CC(flow)=4 and CC(eventHandlers)=3.
```

Listing 8: P2 process metrics

doso, while these metrics are very simple, they are important to complement others [Cardoso, 2008]. The number of activities in a process metric (NOA) counts the number of basic activities while the number of activities and control-flow elements in a process metric (NOAC) also counts structured activities.

In this evaluation we use two variants of our use case scenario. In the first one, the *when-then* construct has a condition with two context variables updated through the request/reply interaction, as we can see in Figure 6. We compare it with the standard WS-BPEL activities that we can use to obtain the same behavior: an *onAlarm* event handler with a *if* activity to evaluate a condition similar to the *when-then* condition, as illustrated in Figure 1. This WS-BPEL standard process P1 has the metrics we present in Listing 7.

By using the *when-then* construct, the CFC and the CC metrics decrease 3 and 4 units, respectively (the bold values). These values are the CFC and the CC of three initial tasks of the *onAlarm* sequence. The NOA and NOAC metrics have a decrease of 2 and 3 units, respectively.

Considering n the number of context variables of this type we use in the *when-then* condition, these metrics have a linear decrease as follows: the CFC and the NOAC metrics decrease $n+1$, NOA decreases n, and CC decreases $n+2$.

In the other variant of our use case scenario, the *when-then* construct has a condition with two context variables updated through the publish/subscribe interaction. We compare it with the standard WS-BPEL process (P2) illustrated in Figure 2. This process P2 has the metrics we present in Listing 8. By using the *when-then* construct, if we consider that the *if-then* branch only has a basic activity (with CFC and CC equal to one) CFC decreases 19 units, CC decreases 19 units (the bold values), NOA decreases 7 units, and NOAC decreases 13 units.

Generalizing these values to a *when-then* construct with a condition with n context variables we get the following results: CFC decreases $n!(2n)+n\times n\times 3-1$, CC decreases $8+2n+3+4(n-1)$, NOA decreases $2n+2(n-1)+1$, and NOAC decreases $1+3n+4(n-1)+2$. Even if we use the *when-then* construct without context variables, our approach avoids busy waiting.

Highly complex processes are error prone, more difficult to understand and to maintain [Mao, 2010]. Decreasing WS-BPEL size and complexity increases readability [Cardoso, 2008].

7.3 Performance evaluation

To make the performance tests, we used computers with the following characteristics: CPU - Intel QuadCore 2.33 GHz, RAM - 6 GB, and Operating System - Windows 7. The WS-BPEL process we used in these tests has a context variable that is updated using the publish/subscribe interaction, as well as a *when-then* construct with a `temperature` $>$ `35` condition. As our implementation has no impact on the business process execution engine, we focus our performance tests in the auxiliary web service we used to generate the standard WS-BPEL process definition. We compared the two approaches referred in section 6.2 to detect value changes in context variables: the one that uses *listeners* and the other that adds new activities to the process definition.

We executed several process instances simultaneously to register several *when-then* conditions being processed by our auxiliary web service, so that we could assess its scalability. To determine the response time of the auxiliary web service we used *logs*: we calculate the difference between the time when the process instance changes the variable value, and the time when the *when-then* construct is unlocked. Figure 7 presents the average values for the response times in milliseconds (ms). As we can see, the response time increases linearly for both approaches as more instances are executed. The second approach (the one that does not use *listeners*) has an higher response time as it exchanges more messages. However, it also presents a linear growth.

8 Conclusion

IoT information is becoming widely used by organizations in their business processes as a competitive advantage. However, to use it, process modelers have to scatter process definitions with additional operations to interact with IoT-enabled technologies such as sensors.

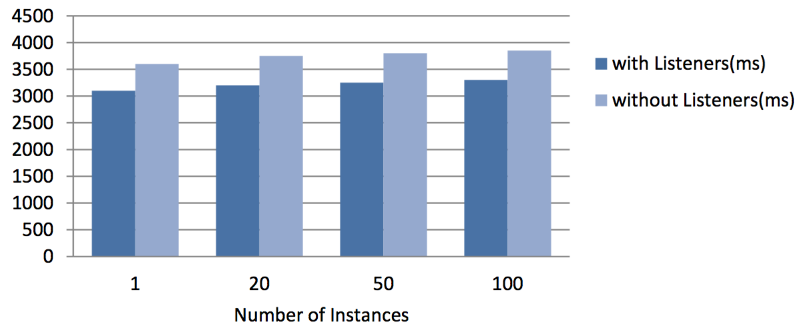

Figure 7: Performance tests results: response times vs transformation approaches vs number of process instances

The work we present in this paper aims at simplifying the access to IoT information within WS-BPEL processes. Through a WS-BPEL extension, processes can include context variables, whose values are updated automatically and synchronously/asynchronously: the extension is responsible for the operations required to perform the communication between process instances and sensors. In addition, the WS-BPEL extension we propose also includes a *when-then* construct that process modelers can use to define expected exceptions, using conditions with context variables.

We realize the WS-BPEL extension through a language transformation approach. As it adds new activities to process definitions, processes that are executed do not match exactly the process the modeler defined. Nevertheless, the resulting process behaves as expected by the modeler and is independent from the execution engine.

We also show how this extension avoids the increase of WS-BPEL code size and complexity and the scatter of additional code through the process definition, letting process modelers to focus on the modeling of main business logic.

Future work includes studying how to support similar functionalities in BPMN, as well as evaluating the usability of the proposed extension.

Acknowledgments

This work was supported by the portuguese Foundation for Science and Technology (FCT) through the PATI project (PTDC/EIAEIA/103751/2008) and the LaSIGE Strategic Project, ref. PEst-OE/EEI/UI0408/2014.

References

[Apache Muse, 2014] Apache Muse (2014). URL: `http://ws.apache.org/muse/` Page visited on February 10th, 2014.

[Apache ODE, 2014] Apache ODE (2014). URL: http://ode.apache.org/ Page visited on February 10th, 2014.

[Apache Tomcat, 2014] Apache Tomcat (2014). URL: http://tomcat.apache.org/ Page visited on February 10th, 2014.

[BPEL Designer Project, 2014] BPEL Designer Project (2014). URL: http://www.eclipse.org/bpel/ Page visited on February 10th, 2014.

[Cardoso, 2007] Cardoso, J. (2007). Complexity analysis of bpel web processes. *Software Process: Improvement and Practice*, 12(1):35–49.

[Cardoso, 2008] Cardoso, J. (2008). Business process control-flow complexity: Metric, evaluation, and validation. *International Journal of Web Services Research*, 5(2):49–76.

[Casati et al., 1999] Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G. (1999). Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems (TODS)*, 24(3):405–451.

[Clark et al., 2007] Clark, J., Deach, S., and Kay, M. (2007). XSL transformations. Saxonica, Adobe.

[Domingos et al., 2013] Domingos, D., Martins, F., and Cândido, C. (2013). Internet of things aware WS-BPEL business process. In *Proceedings of ICEIS'13*, pages 505–512.

[Eclipse IDE, 2014] Eclipse IDE (2014). Eclipse IDE for Java EE Developers. URL: http://www.eclipse.org/ Page visited on February 10th, 2014.

[George, 2008] George, A. (2008). Providing context in WS-BPEL processes. Master's thesis, Electrical and Computer Engineering, University of Waterloo.

[George and Ward, 2008] George, A. A. and Ward, P. A. S. (2008). An architecture for providing context in WS-BPEL processes. In *Proceedings of CASCON'08*, pages 22:289–22:303. ACM.

[Informatica, 2014] Informatica (2014). Active BPEL. http://www.activevos.com/learn/bpel.

[Jedermann and Lang, 2008] Jedermann, R. and Lang, W. (2008). The benefits of embedded intelligence - tasks and applications for ubiquitous computing in logistics. In *Proceedings of IoT'08*, pages 105–122.

[Karastoyanova et al., 2005] Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., and Buchmann, A. (2005). Extending BPEL for run time adaptability. In *Proceedings of EDOC'05*, pages 15–26. IEEE Computer Society.

[Kopp et al., 2011] Kopp, O., Grlach, K., Karastoyanova, D., Leymann, F., Reiter, M., Schumm, D., Sonntag, M., Strauch, S., Unger, T., Wieland, M., and Khalaf, R. (2011). A classification of BPEL extensions. *Journal of Systems Integration*, 2(4):3–28.

[Krizevnik and Juric, 2012] Krizevnik, M. and Juric, M. B. (2012). Data-bound variables for WS-BPEL executable processes. *Computer Languages, Systems & Structures*, 38(4):279–299.

[Liu et al., 2007] Liu, A., Li, Q., Huang, L., and Xiao, M. (2007). A declarative approach to enhancing the reliability of bpel processes. In *Proceedings of ICWS'07*, pages 272–279. IEEE.

[Mao, 2010] Mao, C. (2010). Control and data complexity metrics for web service compositions. In *Proceedings of QSIC'10*, pages 349–352. IEEE.

[Mateo et al., 2012] Mateo, J. A., Valero, V., and Díaz, G. (2012). BPEL-RF: A formal framework for BPEL orchestrations integrating distributed resources. *CoRR*, abs/1203.1760.

[Meyer et al., 2013] Meyer, S., Ruppen, A., and Magerkurth, C. (2013). Internet of things-aware process modeling: Integrating IoT devices as business process resources. In *Advanced Information Systems Engineering*, pages 84–98. Springer.

[Meyer et al., 2012] Meyer, S., Sperner, K., Magerkurth, C., Debortoli, S., and Thoma, M. (2012). IoT-A Project Deliverable D2.2: Concepts for Modelling IoT-Aware Processes. http://www.iot-a.eu/public/public-documents/documents-1/1/1/copy4_of_d1.2/at_download/file. Visited page on February, 2014.

[Muketha et al., 2010] Muketha, G., Ghani, A., Selamat, M., and Atan, R. (2010). A survey of business process complexity metrics. *Information Technology Journal*, 9(7):1336–1344.

[OASIS, 2006a] OASIS (2006a). Web services notification (WS-Notification) version 1.3. OASIS. URL: `https://www.oasis-open.org/committees/wsn/`.

[OASIS, 2006b] OASIS (2006b). Web services resource properties version 1.2. OASIS. URL: `http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf`.

[OASIS, 2007] OASIS (2007). Web services business process execution language version 2.0. Technical report, Organization for the Advancement of Structured Information Standards.

[OMG, 2011] OMG (2011). Business process model and notation (BPMN), version 2.0. Technical report, Object Management Group.

[Predic and Stojanovic, 2012] Predic, B. and Stojanovic, D. (2012). Localized processing and analysis of accelerometer data in detecting traffic events and driver behaviour. *Journal of Universal Computer Science*, 18(9):1152–1176.

[Ranganathan and McFaddin, 2004] Ranganathan, A. and McFaddin, S. (2004). Using workflows to coordinate web services in pervasive computing environments. In *Proceedings of ICWS'04*, pages 288–. IEEE Computer Society.

[Reichert and Rinderle, 2006] Reichert, M. and Rinderle, S. (2006). On design principles for realizing adaptive service flows with BPEL. In *Proceedings of EMISA'06*, pages 133–146. Koellen-Verlag.

[Saxon Home Edition, 2014] Saxon Home Edition (2014). URL: `http://saxon.sourceforge.net/` Page visited on February 10th, 2014.

[Wieland et al., 2009] Wieland, M., Gorlach, K., Schumm, D., and Leymann, F. (2009). Towards reference passing in web service and workflow-based applications. In *Proceedings of EDOC'09*, pages 109–118. IEEE.

[Wieland et al., 2007] Wieland, M., Kopp, O., Nicklas, D., and Leymann, F. (2007). Towards context-aware workflows. In *Proceedings of CAISE'07*. Citeseer.

[Yu and Su, 2009] Yu, L. and Su, S. (2009). Adopting context awareness in service composition. In *Proceedings of Internetware'09*, pages 11:1–11:10. ACM.

[Zeng et al., 2011] Zeng, D., Guo, S., and Cheng, Z. (2011). The web of things: A survey (invited paper). *Journal of Communications*, 6(6).

[Zeng et al., 2005] Zeng, L., Lei, H., Jeng, J.-j., Chung, J.-Y., and Benatallah, B. (2005). Policy-driven exception-management for composite web services. In *Proceedings of CEC'05*, pages 355–363. IEEE.

[Zhang et al., 2012] Zhang, D., Ning, H., Xu, K. S., Lin, F., and Yang, L. T. (2012). Internet of things. *Journal of Universal Computer Science*, 18(9):1069–1071.