

Analysis of the WS-BPEL 2.0 standard using standard-driven implementation

Tim Hallwyl¹, Fritz Henglein¹, and Thomas Hildebrandt²

¹ Department of Computer Science, University of Copenhagen
{hallwyl,henglein}@diku.dk

² IT University of Copenhagen
hilde@itu.dk

Abstract. We present a systematic study of the OASIS WS-BPEL 2.0 standard (henceforth simply called BPEL) based on two complementary methods: the process of constructing a new high-level BPEL implementation driven by the structure of the standard, and an empirical evaluation of existing interpretations of the standard reflected in five widely available BPEL-implementations, both commercial and open source.

In doing so we uncover a number of new ambiguities. Most notably, BPEL’s integration of XPath 1.0, the data access component of BPEL, turns out to be inconsistent with the XPath standard itself, which is evidenced by substantially differing results produced by existing implementations on test cases constructed to exercise their interpretation.

The core concepts in BPEL have been formalized and analyzed successfully previously. We believe this to be the first study of the complete BPEL standard, however, that investigates its integration with other standards, notably XPath. Given BPEL’s design goal of being platform-independent the inconsistencies are arguably a serious concern since they cannot be attributed to the quality of any particular implementation.

Beepell, the BPEL engine and visual execution environment implemented by the first author in Java that has resulted from this effort is open source and freely available for educational, research and application purposes.

1 Introduction

The *Web Services Business Process Execution Language Version 2.0 (WS-BPEL)* [1] is a language for the specification of executable and abstract business processes. It exports and imports functionality using Web Service interfaces exclusively to ensure that processes be reusable, deployable “in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them.” [1, Section 1]. In particular, BPEL is to ensure *portability* across different BPEL execution engines, both existing and future ones.

For this reason, WS-BPEL is based on the *Extensible Markup Language (XML)* [2], which is designed to provide portability through a platform-neutral encoding of semi-structured data, and a number of web service standards that build on XML. The *WS-BPEL specification* consists of a number of XML schemas

and namespaces defining the syntax of the language; and a *principal prose document* [1], which we henceforth informally refer to as the (*BPEL*) *standard*, specifying its semantics.

The standard, being written in natural language, carries the obvious danger of containing ambiguities, be it inconsistent requirements or underspecifications. An inconsistency entails that at least one of a number of requirements in the standard cannot be implemented (but which one of several?); and an underspecification requires choosing among several possibilities by the implementor (but which one?). In the first category we include situations where a literal reading in the standard may be clear enough, but ostensibly inconsistent with underlying intentions.

1.1 Goal and method

Our goal has been to develop an implementation-driven method that scales to the complete BPEL standard for identifying practicality relevant ambiguities in BPEL, complementing analyses of BPEL's process-theoretic core based on semantic formalizations.

But what is an ambiguity in *practice*? What may be a literal underspecification may have a canonical implied resolution. Likewise, what may be *a priori* logically inconsistent may have a commonly accepted resolution in practice, as is common in legal reasoning.

We can approach this question empirically: If different BPEL engines exhibit different observable behavior this may indicate a potential ambiguity in the standard. But how do we find test cases that bring forth such differences? Not only are they bound to be difficult to find in practice where BPEL processes typically execute only on a particular platform. A difference of behavior under different BPEL engines may also be due to a buggy or incomplete BPEL implementation or to an intended implementation freedom in the standard, neither of which the standard can be blamed for.

For this reason we approach our goal using two complementary methods: a *standard-driven* implementation of a BPEL engine for the *complete* BPEL standard, including full process interaction with partners and implementation of referenced standards; and a comparative evaluation of the interpretation of test cases exercising potential ambiguities by existing BPEL engines.

Our implementation is standard-driven in the sense that each part of the standard is sought to be reflected as directly and at as high a level as possible in the source code. This approach is inspired by Karpf's *Isomorphism Principle* for formalization of legal systems [3, 4]. The purpose of developing an implementation whose design follows the structure of the standard is two-fold: It is to drive the systematic analysis of the standard so as to uncover potential inconsistencies and underspecifications; and it is to make the implementation and the standard easily interrelatable.

Once a potential ambiguity is located in this process, a paradigmatic test case is constructed and submitted to existing BPEL engines. If they produce

different results we take this as supporting evidence that we have uncovered a practically relevant ambiguity.

Since we use the engines to analyze the standard, not to compare them with respect to each other for other purposes such as completeness, performance, integration support, etc., it is not significant which particular versions we use. A difference in behavior is supporting evidence for a potential ambiguity in the standard, even if implementors subsequently communicate with each other and agree on a common resolution.

1.2 Contributions

The work presented in this paper provides a systematic semantic study of the BPEL standard which

- is based on the development of a new open-source, standard-driven Java implementation
- goes beyond the core of BPEL and includes other standards referenced, notably the integration of XPath,
- supports visual, interactive execution of processes,
- reveals new practically significant inconsistencies in the BPEL standard, primarily related to XPath integration, and
- validates the inconsistencies empirically by obtaining mutually different results when run on test cases exercising the issues on five widely available, existing BPEL implementations.

The key features regarding the control flow of BPEL processes are already well investigated using formalizations based on Abstract State Machines [5, 6], Petri nets [7] and process calculus and labelled transition semantics [8]. However, none of the formalizations consider the integration of referenced standards. We believe that our work is the first systematic semantic study of the whole standard that goes beyond the core of BPEL and includes other standards used, notably XPath. This task was made feasible by our choice to use a high-level, standard-driven implementation of the BPEL standard rather than an abstract, mathematical formalization. The use of existing rich libraries and software components facilitates scaling the analysis to included referenced standards, notably XPath, that have previously been abstracted away or modeled independently of the standard.

1.3 Overview

In Section 2 we give a brief introduction to the BPEL standard and the standards referenced in it. In Section 3, we present Beepell, our standard-driven BPEL implementation and the five BPEL engines used in our empirical study. Section 4 presents the ambiguities (“issues”) we have encountered during implementation and how they are interpreted in existing implementations. We conclude the paper with a discussion (Section 5) and a review of related work (Section 6).

2 The WS-BPEL 2.0 Standard

BPEL is a language for specifying both abstract and executable business processes modeled as the interaction of a (local) *process* and its (remote) partners. It builds on several XML specifications: WSDL 1.1 [9], XML Schema 1.0 [10, 11], XPath 1.0 [12] and XSLT 1.0 [13]. WSDL messages and XML Schema type definitions provide the data model used by BPEL processes. XPath and XSLT provide support for data manipulation.

Below we briefly describe the description of processes and the referenced standards. We occasionally refer to XML Information Set (InfoSet) [14] and Document Object Model (DOM) [15], but in-depth knowledge of these standards is not required.

2.1 Process Description

The core notion in BPEL is a *process description*. A process description is a composition of *activities*. BPEL defines two sets of activities: structured and basic. The set of structured activities controls the flow of the process, while the basic activities mainly perform data manipulation and web service operations.

A process description is instantiated by the arrival of a message. The content of that message is stored in a variable of the *process instance*. The instance then executes as described in the description until it terminates when its main activity completes.

2.2 Referenced Standards

The BPEL standard references a number of existing standards. They are introduced below.

XML Schema BPEL adopts the data model and type system of XML Schema. In particular, all variables and expressions in BPEL are explicitly typed using XML Schema types. Variables declared within a process description can be of XML Schema simple type, complex type or element, or of WSDL message type.

WSDL The web services used in a process description are defined using WSDL. In WSDL web service interfaces are defined in terms of ports, operations and messages. As message definitions in WSDL are based on XML Schema types, the integration of messages as a variable type in BPEL fits with XML Schema as the type system.

BPEL extends WSDL with the notion of *partner links* and *variable properties*. Partner links define the relationship between two partners, essentially by which port that one partner must publish to another. Variable properties allows one to define a named and typed property and the locations (aliases) within other types where the property is found. For example, a property ‘order-number’ may be defined as an `xsd:integer` with locations within `foo:order`, `foo:confirmation`

and `foo:invoice`. The declaration of variable properties relies on XPath to describe the location of the property within a complex type, element or message definition.

XPath BPEL does not provide its own language accessing (selecting parts of) data stored in variables. Instead, it relies on an integration of XPath as its data manipulation language. Note that XPath has a data model that is not XML Schema aware, which makes this integration nontrivial.

XSLT BPEL does not provide its own language for defining functions on (transformations of) data. Instead, it employs XSLT in combination with XPath to support user-definable functions [1, 8.4]. As with XPath, XSLT is not XML Schema aware.

3 BPEL implementations

In this section we briefly describe our standard-driven implementation and the empirical study involving five existing BPEL engines.

3.1 Beepell: A Standard-driven implementation

To gain insight into the standard and, at the same time, perform a thorough investigation of it, we have constructed a full implementation capable of executing any standard compliant process description.

Our implementation follows an object-oriented approach in order to map the conceptual descriptions in the BPEL standard directly into the source code. This, along with a graphical user interface (see Fig. 1) showing the real time process execution (see Figure 1), is intended to make the implementation transparent and easily disputable for anyone who has a different interpretation of the standard than we do.

A thorough description of the design and implementation of Beepell is outside the scope of this paper. Its source code and executables are open-source available. They can be downloaded from <https://sourceforge.net/projects/ws-bpel/>.

3.2 Comparative Empirical Study

We have included five implementations, besides our own, in the empirical study. They are listed in Table 1. Our own implementation is listed as ‘beepell’ in the following.

The studies were carried out at the end of first quarter of 2008. While the implementations may have changed since this study was carried out, this only confirms the insufficiency of the BPEL standard: Even if the implementations converge on a consistent interpretation, the differences on the way show that such a consistency is not provided by the standard itself.

Fig. 1. A screen shot of the graphical representation of process execution provided by our implementation.

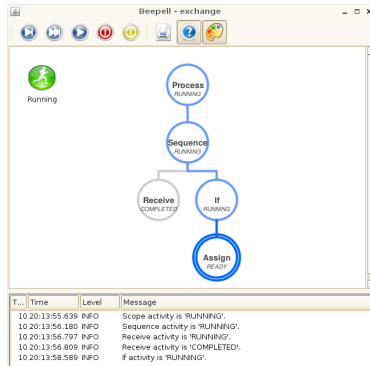


Table 1. Implementations includes in our empirical study

Implementation	Version
BEA AquaLogic	6.0
Apache's Orchestration Director Engine	1.2
Sun BPEL Service Engine	6.0.110
ActiveBPEL Community Edition Engine	5.0 M1
JBoss Application Server jBPM	1.1.GA

4 Results

As most of the issues we encountered are related to the integration of XPath we shortly introduce XPath and the basic requirements imposed by the BPEL standard. We then present five issues, each with a presentation of the related requirements, analysis of the issue, the resolution we implemented and a test case exploring the other implementations.

4.1 XPath

XPath is a side effect free language designed to address nodes within an XML document tree. It uses a path notation to navigate through the document tree, selecting a node. For example the expression `‘/order/items/item[2]/quantity’` selects the quantity element of the second item in the items sequence within the order element. XPath also provides the basic facilities for manipulation of text strings, along with functions and operators for numeric and Boolean expressions.

An XPath expression returns an XPath object. There are just four types of objects in XPath: String, Number, Boolean and Node-Set. The String object consists of zero or more characters. A Number represents a floating-point number³. An object of type Boolean can have one of two values, *true* or *false*. A

³ A 64-bit format IEEE 754 number, which includes the special value Not-a-Number.

Node-Set is an unordered collection of nodes without duplicates. In the XPath data model, there are seven types of nodes including element, attribute and text nodes. [12]

All XPath expressions are evaluated with respect to a *context* consisting of the following five parts: the context node, the context position and size, a set of variable bindings, a function library and a set of namespace declarations.

The BPEL variables in the scope of the expression are bound as XPath variables, with the exception of expressions used in join conditions and variable property aliases. The variables of join conditions are bound to the synchronization links in scope while expressions in variable property aliases do not have any variables bound.

In addition to the ‘Core Function Library’ provided by XPath, the BPEL standard specifies two XPath functions: `getVariableProperty` to retrieve a variable property value and `doXsltTransform` to perform an XSL transformation.

XPath is not XML Schema aware. This means that any value selected within the document tree is represented as either a Node-Set or a String object. Selecting the value of a `xsd:boolean` typed attribute node, for example, yields a String object with the literal representation of the Boolean value. However, when the selected value is part of a Boolean or numeric expression it is implicitly converted into an XPath Boolean or Number object, respectively.

The BPEL standard distinguishes between *queries* and *expressions*. Both are XPath expressions, however. The main difference is that a query is evaluated with a BPEL variable value as the context node, while expressions are evaluated with an empty document as the context node.

Queries are used in copy operations to access a specific element or an attribute within a variable value. In variable property aliases, queries are used to point out where the property value is found, within a type, element or message definition.

Expressions are used in conditional constructs, for example in If- and While-activities, and to retrieve values that may be calculated, for example dates and durations for Wait-activities. In the BPEL standard, expressions are classified by the type of value they should return, as shown in Table 2. General expressions are used in assignments.

Table 2. Types of expressions, their return type and applied conversion

Expression Type	Return Type	Conversion
Boolean expressions	<code>xsd:boolean</code>	<code>boolean(object)</code>
Deadline expressions	<code>xsd:date</code> and <code>xsd:dateTime</code>	<code>string(object)</code>
Duration expressions	<code>xsd:duration</code>	<code>string(object)</code>
Unsigned Integer expressions	<code>xsd:unsignedInt</code>	<code>number(object)</code>
General expressions	<code>any</code>	<code>none</code>

4.2 Issues uncovered

Issue 1: XPath Context The BPEL standard prescribes that queries must have either “node-list or object” as the context node. However, XPath accept neither a node-list nor an object as the context node: according to the XPath specification the context node is a single node [12].

In the case of the node-list, the standard requires it to be “a node-list containing a single node” [1, Section 8.2.6]. This issue is easy to overcome, using the single node in the list as the context node.

If the type is a simple type, the context node MUST point to the XPath object specified in section 8.2.2 [1, Section 8.2.6]

The quote above tells us that the standard insists on using an XPath object as the context node when the query of a simple typed variable property alias is evaluated. This is clearly not consistent with the XPath specification. The question is how to implement an impossible requirement.

We constructed a test case in an attempt to reveal how other implementations go about this. But it is difficult to explore because we cannot select the context node as an object. We try to explore it with an assignment operation, copying from a `xsd:boolean` variable using a query: ‘boolean(.)’ — the dot is selecting the current node.

If the Boolean variable is passed as an XPath object, then it should return the Boolean value of the variable: *false*. If it is passed as a node, the Boolean value *true* is expected, disregarding the variable’s value. The last option is to fail.

Table 3 shows the results of this test case: Three of the implementations seem to have solved this issue in some way while two fail. Our own simply passes the value as a text node.

Table 3. Attempt to use an object as the context node

Implementation	Result Logged
Beepell	true
Apache ODE	false
JBoss jbp	failed selection is not a node: true
Sun BPEL SE (GlassFish)	false
Active BPEL	false
BEA AquaLogic SOA Suite	failed could not be successfully executed

Issue 2: Return Values The return types listed for the typed expressions in Table 2 are “conforming” types. As XPath is not aware of XML Schema types, the value returned is only required to *conform* with the specified type. A deadline expression, for example, returns a Text node conforming with `xsd:date`.

All typed BPEL expressions return a sequence of character information items (CII) in the InfoSet model, equivalent to a Text node in DOM. However, BPEL requires specific conversion methods to be applied, as listed in the ‘Conversion’ column of Table 2.

```
<foo:account active="false">
  <foo:name>Leased Equipment</foo:name>
</foo:account>
```

When selecting a Boolean value—for example from the `xsd:boolean` typed ‘active’ attribute in the listing above—then XPath selects a Text node. Applying the `boolean` XPath function is required by BPEL because XPath does not know of the XML Schema types. This will, however, convert a selected value of ‘false’ into a Boolean value *true*.

The `boolean` function does, according to the XPath specification, convert all non-zero length Text nodes or String objects to *true*. Only empty strings are converted to *false*.

```
<bpel:if>
  <bpel:condition>${account}/@active</bpel:condition>
  ...
</bpel:if>
```

As a consequence the activity contained in the If-activity above is always executed, as the condition always returns *true*. Obviously, this has a significant impact on how the language can be used. The question is if implementors choose to implement another—more practical—conversion.

To explore this, we try evaluating a Boolean expression that evaluates to an XPath String object of value ‘false’. Using the conversion methods required by the BPEL standard, this should return the Boolean value *true*. Table 4 below shows the results.

As it turns out that half of the tested implementations return *true* and the other half *false*, we are forced to conclude that portability of process descriptions is severely reduced by this issue.

The implementations that return *true* are consistent with the BPEL standard and the definition of the XPath `boolean` function. These implementations make it rather difficult to base Boolean expressions on `xsd:boolean` values selected within variables, however.

Issue 3: getVariableProperty The `getVariableProperty` function is defined as follows:

The return value of this function is calculated by applying the appropriate `<vprop:propertyAlias>` for the requested property to the current value of the submitted variable. [1, Section 8.3]

Table 4. Implicit conversion of Boolean expressions

Implementation	Result
Beepell	true
Apache ODE	true
JBoss jbpn	false
Sun BPEL SE (GlassFish)	true
Active BPEL	false
BEA AquaLogic SOA Suite	false

This gives a good idea of the intention of the function, but leaves us with the understanding that the XPath function should return the same as when a variable property is referred directly in an assignment: a single information item other than CII, or a sequence of zero or more CIIs.

There is nothing invalid about returning a sequence of CIIs; in XPath this will be mapped to a String object. It does, however, impose some problems, having all simple type values represented as XPath String object.

```
<bpel:if>
  <condition>
    bpel:getVariableProperty('shipRequest', 'props:shipComplete')
  </condition>
  ...
</bpel:if>
```

This is especially clear in simple Boolean expressions, as shown in the above example. A conversion using the XPath `boolean` function will implicitly be added [1, Section 8.3]. However, if the function returns a String object, for example 'false', the conversion will return *true* — a string is *true* if and only if its length is non-zero. Thus, the expression in the above example will always return *true*.

There are two hints that suggest an implicit conversion of variable properties into proper XPath objects. The first hint is that the method signature returns an object. The second hint is that the example above is from the general examples section in the BPEL standard [1, Section 15.1.3]. The `props:shipComplete` property is of `xsd:boolean` type.

We suspect that the same conversion method as used with variables in XPath expressions [1, Section 8.2.2], is intended — though it is not specified.

In a test case we use the `getVariableProperty` function to retrieve a *false* Boolean value, as part of a Boolean expression. The results are listed in Table 5. Besides our own implementation, it seems only Apache ODE and Active BPEL supports this function. They both return *true* meaning that they agree with us on the strict interpretation that `getVariableProperty` does not apply any implicit conversion.

Issue 4: Use of Functions in Join Conditions According to the section on static analysis in the BPEL standard, a join condition expression must be

Table 5. Retrieve a Boolean ‘false’ using `getVariableProperty`

Implementation	Result Logged
Beepell	true
Apache ODE	true
JBoss jbpn	failed No such function
Sun BPEL SE (GlassFish)	N/A
Active BPEL	true
BEA AquaLogic SOA Suite	failed XPathSyntaxException: Unexpected '('

constructed using only Boolean operators and the status values of the activity’s incoming links [1, SA00073]. Section 8.2.5 in the standard, on the XPath context for join conditions, does however allow access to XPath functions.

In our implementation, we allow XPath functions in join conditions. However, this is only because we do not implement static analysis.

We constructed a test case using core XPath functions within a join condition to investigate if other implementations are allowing this. Table 6 below shows the results: The BEA and Sun implementations do not support synchronization links. Active BPEL rejected the process description at deployment while Apache and JBoss allowed it.

Table 6. Are XPath functions allowed in join conditions

Implementation	Result	Logged
Beepell	Allowed	
Apache ODE	Allowed	
JBoss jbpn	Allowed	
Sun BPEL SE (GlassFish)	N/A	
Active BPEL	Rejected	Illegal function call [floor] in join condition.
BEA AquaLogic SOA Suite	N/A	Transitions from activity ... are not allowed.

Issue 5: ‘Identical’ Values in Correlation Sets A *correlation set* is initiated with values from a message. It is then used to route incoming messages to the right process instance.

The correlation semantics is based on two constraints that must be observed: the initiation and consistency constraints. The consistency constraint is defined as follows:

After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set [1, Section 9.2]

On the one hand it is easy to understand the intentions, but on the other hand the concept of ‘identical value’ is a bit vague.

Because correlation only uses simple type properties, the result cannot be anything but a Text node (a sequence of CII in the Infoset model).

Since it is the same property that is applied to different messages, the type is per definition identical. But the BPEL standard does not offer any guidelines on how to decide if two values are identical. Since Infoset is used to explain the concept of ‘value’, we could assume that identical simple type values have identical CII sequences.

However, comparing the textual values directly may not be the intention. For example, a property of type `xsd:decimal` may retrieve two values using different aliases, such as ‘42.10’ and ‘42.1’. Are the values identical?

The standard does in general represent a simple type value as an InfoSet CII sequence, and in that sense the answer must be ‘no’—they do not have identical Infoset CII sequences. However, it seems reasonable to expect that decimal simple type values are compared by their numerical value, especially since the properties are typed.

This issue regards a wide range of data types. Another example is the Boolean value *false* that may be expressed as either ‘false’ or ‘0’. In our implementation we follow the implications of the BPEL standard and compare the textual values, ignoring the data type altogether.

Testing how this is implemented by others is a time consuming task because we need to manipulate the messages that are meant to correlate. For this reason we have set up only one test case comparing our implementation (beepell) with ActiveBPEL.

Table 7 below shows the test results of an attempt to correlate with two properties, a decimal value ‘42.10’ and a Boolean value ‘false’. ‘Yes’ means that a message with the literal values specified was routed to the instance ‘No’ means that the message did match the correlation set.

While our implementation (beepell) strictly requires identical CII sequences, ActiveBPEL does seem to compare Boolean values on their value rather than their representation, but not with the decimal numbers. We have informally tried with ‘true’ instead of ‘false’, and this gave the same results. The results emphasize the need for canonicalization of correlation values when designing processes.

Table 7. Attempt to correlate messages with ‘identical’ values.

Decimal	Boolean	beepell	ActiveBPEL
42.1	false	No	No
42.1	0	No	No
42.10	false	Yes	Yes
42.10	0	No	Yes
42.100	false	No	No
42.100	0	No	No

5 Discussion

We have presented a systematic semantic study of the BPEL standard that goes beyond the core of BPEL and includes other standards referenced in the BPEL standard, notably XPath. The process (not just the result) of constructing the implementation from the standard helped identifying potential ambiguities in the standard outside the process-theoretic core of BPEL. By performing a comparative analysis, constructing and applying a set of test cases to our implementation and five widely available implementations of BPEL, we have provided empirical evidence that the issues identified indeed give rise to inconsistencies between the different implementations in practice.

A key finding is that the source of what we consider to be the most significant inconsistencies uncovered here is attributable to the XPath and BPEL standards having different data models and type systems for XML documents and the lack of an explicit consistent mapping of one data model into the other.

The choice of using a high-level Java implementation has made it feasible to cover the entire (mandatory part of the) language specification including referenced standards such as XPath, which have been left out of previous formalizations based on mathematical models. By using Java as our “formalization” language we forfeit *mathematical* reasoning. Also, there may be ambiguities or under-specifications inherited from Java. For instance, we rely on the Java thread scheduling for handling execution of concurrent flows. We are not aware of any mathematical model language which at the same time supports a high-level, standard-driven representation of the *complete* standard and also allows execution of processes, however. An intermediate step towards this would be a mathematical, executable model language with support for integration of XPath as an external component. As part of the CosmoBiz project⁴ we are presently working along this direction, replacing parts of the Java implementation with an executable formalization based on the bigraph formalism [16] and implementing an engine for executing bigraphs that allow integration of XPath as expression language. As demonstrated in [17, 18] bigraphs are able to represent XML data and BPEL processes very directly, making the formalism promising for a standard-driven formalization.

6 Related Work

The work reported here is based on the first author’s Master’s thesis [19].

The work in [5–8] has similar aims as ours, namely uncovering ambiguous or invalid requirements in the language specification. Fahland et al [5, 6] provide a complete formal model of the architecture and semantics using the Abstract State Machine (ASM) formalism and propose to check implementations against the formalization using model based testing methods. As in our approach, the need for a formalization at a level of abstraction described in the informal specification is emphasized, for which ASM indeed appears very suitable. Lohmann [7]

⁴ See <http://www.cosmobiz.org>.

provides a feature complete semantics in high-level Petri Net claiming to cover all data and control flow aspects of WS-BPEL 2.0. The formalization does not take into account the referenced standards nor invocation of services, however. Finally, Lapidula et al. [8] provide a formalization of central aspects (but not all) of the standard in a BPEL-like process calculus, *Blite*, invented for the purpose. Interestingly, the authors identify a number of issues related to the invocation of services, concurrent execution and correlation of messages and also demonstrate by an empirical test of three commercial and open source engines that they indeed behave differently with respect to these issues. By not treating the integration of XPath and in particular the evaluation of expressions all of these approaches thereby fail to identify the issues revealed in the present study. This also means that, even though the Petri Net semantics is executable, none of the formalizations can be used directly for executing BPEL processes.

A number of other authors have contributed significantly to the understanding of BPEL by providing formalizations. Rather than aiming at an analysis of the standard, however, their aim has been to identify the essential principles of business processes and/or provide means for automatic verification and analysis of key issues, abstracting from other issues.

Acknowledgements

This work has been partially supported by the Danish National Advanced Technology Foundation under Project *3gERP* (3d generation Enterprise Resource Planning systems, 3gERP.org), by the Danish Research Council for Technology and Production under Project *Cosmobiz* (Grant no. 274-06-0415, cosmobiz.org) and by Sirius IT (siriusit.com).

References

1. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Golland, Y., Guizar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WS-BPEL) TC (April 2007)
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, W3C (1999)
3. Karpf, J.: Quality Assurance of Legal Expert Systems. *Jurimatics* **8** (1989) Copenhagen Business School.
4. Bench-Capon, T.J.M., Coenen, F.P.: Isomorphism and legal knowledge based systems. *Artificial Intelligence and Law* **1** (1992) 65–86
5. Fahland, D.: Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. *Informatik-Berichte 190*, Humboldt-Universität zu Berlin (September 2005)
6. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: The negative Control Flow. In Beauquier, D., Börger, E., Slissenko, A., eds.: *Proceedings of the 12th*

- International Workshop on Abstract State Machines (ASM'05), Paris XII (March 2005) 131–151
7. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. *Informatik-Berichte* 212, Humboldt-Universität zu Berlin (August 2007)
 8. Lapadula, A., Pugliese, R., Tiezzi, F.: A formal account of WS-BPEL. In: Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08). Volume 5052 of Lecture Notes in Computer Science (LNCS)., Springer (2008) 199–215
 9. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3C Note, W3C (2001)
 10. Thompson, H., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures Second Edition. W3C Recommendation, W3C (2004)
 11. Biron, P., Permanente, K., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, W3C (2004)
 12. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C Recommendation, W3C (1999)
 13. Clark, J.: XSL Transformations (XSLT) Version 1.0. W3C Recommendation, W3C (1999)
 14. Cowan, J., Tobin, R.: XML Information Set (Second Edition). W3C Recommendation, W3C (2004)
 15. Hors, A.L., Hégarret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, W3C (2004)
 16. Milner, R.: *The Space and Motion of Communicating Agents*. Cambridge University Press (March 2009)
 17. Hildebrandt, T., Niss, H., Olsen, M.: Formalising business process execution with bigraphs and Reactive XML. In: COORDINATION'06. Volume 4038 of LNCS., Springer (2006) 113–129
 18. Bundgaard, M., Glenstrup, A.J., Hildebrandt, T., Højsgaard, E., Niss, H.: Formalizing higher-order mobile embedded business processes with binding bigraphs. In: Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION'08). Volume 5052 of Lecture Notes in Computer Science (LNCS)., Springer (2008) 83–99
 19. Hallwyl, T.: Evaluating the BPEL standard specification. Master's thesis, Department of Computer Science, University of Copenhagen (May 2008)