Edit Distance-based Pattern Support Assessment of Orchestration Languages

Jörg Lenhard, Andreas Schönberger, and Guido Wirtz

Distributed and Mobile Systems Group, University of Bamberg, Germany
{joerg.lenhard,andreas.schoenberger,guido.wirtz}@uni-bamberg.de

Abstract. Orchestration languages are of paramount importance when implementing business processes based on services. Several languages for specifying Web Services-based orchestrations are available today. Examples are the Web Services Business Process Execution Language or Windows Workflow. Patterns for process-aware information systems have frequently been used to assess such languages. Various studies discuss the degree of support such languages provide for certain sets of patterns. However, the traditional trivalent support measure is limited in terms of granularity and selectivity. This paper proposes an edit distance complexity measure that allows to overcome these issues. The applicability of this measure is demonstrated by an analysis of several orchestration languages using four different pattern catalogs.

Keywords: SOA, Pattern, Edit Distance, Orchestration, BPEL, WF

1 Introduction

Today, *service-oriented architectures* (SOAs) form the primary means to create flexible, interoperable and cooperative information systems. In SOAs, business processes are implemented as *composite services* [15]. Such composite services combine calls to existing services by defining control- and data-flow dependencies between the different service invocations. *Orchestrations* have been introduced in [16] as executable process definitions that use Web Services as primitives for interactions.

Several languages for implementing Web Services-based orchestrations are available, such as the *Web Services Business Process Execution Language* (BPEL) [13] or *Windows Workflow* (WF) [3]. Traditional comparisons of such languages predominantly use patterns to assess their expressiveness. *Workflow control-flow patterns* [23] and *service interaction patterns* [2] are of outstanding importance. The more patterns a language supports and the higher the degree of support it provides is, the more expressive it is.

The traditional pattern support measure offers three possible values, i.e., *direct (+)*, *partial (+/-)* or *no direct support (-)* for a pattern [23, p. 50]. These values are calculated based on the number of language constructs (e.g. decision or loop constructs) needed for implementing a pattern. The constructs needed therefore are used to represent how directly a pattern is supported. If a pattern

can be implemented using a single construct, a solution provides direct support. If a combination of two constructs needs to be used, the result is partial support. In all other cases, there is no direct support. This support measure has been used in various studies [1, 5, 6, 8, 17–20, 23–25, 27]. Unfortunately, it comes with a number of problems, so that the calculation of the degree of support provided even is left out in some studies (cf. [14]):

1. Although discussed by some authors, the support measure in its present form does not reveal whether a pattern can be implemented in a language *at all*. Solutions using more than two constructs may exist. However, these are rated as offering *no direct support*. This low level of granularity results in a limited degree of selectivity provided by the support measure. The notion of *no direct support* can easily be misinterpreted to believe that a pattern cannot be implemented in a language at all.
2. Usually, the degree of support is determined by a *conceptual analysis of the set of core constructs* of a language specification. Due to this, the complexity of pattern solutions and the effort required by the implementer of a pattern is not truly captured. An executable implementation of a pattern might require the use of more constructs and other aspects that do not belong to this set, such as variable or correlation set definitions, which an analysis of a specification does not reveal.
3. Only the *use*, but not the *configuration* of constructs is considered in the traditional support measure. This can lead to an inaccurate classification. Assume the default configuration of a construct provides a complete solution to a pattern in one language. In another language, also a single construct is sufficient, but this construct must be configured in several non-trivial steps. Clearly, the first language supports the pattern more directly and the solution is less costly. Nevertheless, both languages achieve the rating of *direct support*.

The contribution of this paper is the definition of a more accurate support measure that offers higher granularity and selectivity. It hence contributes to alleviating the above deficiencies. This measure is based on the *edit distance* concept [11] and operationalized for pattern-based analysis of orchestration languages. The basic idea is to count the number of steps taken by the user of a language when implementing a pattern, as this is more suitable to capture the effort required by the user. The implementation steps we consider are semantically meaningful units of change, specific for orchestration languages. The proposed measure allows to determine the cost associated with the implementation of a pattern in a certain language and unveils whether or not a pattern can be implemented at all.

To verify the applicability of this support measure, we furthermore provide an extensive analysis of the orchestration languages BPEL 2.0 [13], its implementation in the OpenESB BPEL Service Engine (Sun BPEL), and WF in revision 4 [3]. We analyze these languages for their support of four pattern catalogs, the *workflow control-flow patterns* [19, 23], the *service interaction patterns* [2], the *time patterns* [8] and the *patterns for changes in predefined regions* [24]. These catalogs have been used in various studies, e.g. [4, 6, 12, 14, 25, 27], and describe aspects that are of paramount importance for realistic processes. In total, they

consist of 70 different patterns. We calculate the degree of support with our proposed support measure, the traditional trivalent measure, and compare the results. Although partly based on preceding analyses [2, 19, 25, 27], the assessment of the support provided by the languages in the specified revisions for these pattern catalogs is new.

This paper builds upon previous work on assessing pattern support. In [10], we have introduced a uniform method for checking whether a pattern implementation is valid and we have put forward the idea of using an edit distance-based support measure. The approach for assessing the validity of candidate pattern implementations has been reused for this study and is not further discussed. Concerning the edit distance-based support measure, we provide an operationalization for orchestration languages that enables comparability across orchestration languages and we provide an extensive evaluation of our measure in terms of selectivity and applicability by assessing the above four pattern catalogs.

The paper is structured as follows: In the following section we review related work and discuss relevant pattern catalogs and pattern-based analyses. In Sect. 3 we define and demonstrate the use of the edit distance support measure. Thereafter, we present the results of the analysis and discuss their implications. Finally, Sect. 4 concludes the paper with remarks on future work.

2 Related Work

Research on applying patterns for the design and implementation of SOAs and process-aware information systems is very popular, e.g. [12, 28]. However, the work here focuses on applying patterns to measure the effort for creating process models, in particular orchestration models. Therefore, we first identify relevant pattern catalogs and orchestration languages. Then, we discuss existing studies that evaluate the expressiveness of process languages using patterns and finally focus on metrics for measuring pattern support.

There are numerous approaches that present pattern catalogs for assessing process-aware information systems. The *workflow patterns initiative* started this movement with the *control-flow patterns* [19, 23]. They also cover *data patterns* [18], *resource patterns* [17] and *exception handling patterns* [20]. The first pattern catalog that addresses specific requirements of service-oriented processes with a number of important interaction scenarios is the *service interaction patterns* catalog [2]. Similarly, [22] characterizes business functions that are frequently needed in processes as *activity patterns*. [1] defines correlation mechanisms that are used in service interactions in the form of *correlation patterns*. The means for creating process instances are covered by *process instantiation patterns* [5]. Constructs that allow for flexibility of processes in the face of changes to their structure are captured as *change patterns* in [24]. Common time constraint mechanisms are proposed in [8] in the form of *time patterns*.

In the area of orchestration languages, BPEL [13], currently available in revision 2.0, is a widely accepted standard. It is a mainly block-structured language specification that has been implemented by several vendors. As is

common practice in pattern-based analyses [17–19, 23], an implementation of this language needs to be treated as a separate language. One such implementation is the OpenESB BPEL Service Engine.[1] WF [3] is a proprietary orchestration language maintained by Microsoft as a part of the .NET framework. Since April 2010, it is available in revision 4. Compared to previous versions, it has undergone significant changes. WF contains block-structured and graph-oriented elements.

Almost all of the studies that introduce pattern catalogs also analyze the degree of support that varying languages and systems provide for the patterns in focus [1, 2, 5, 8, 17–20, 23, 24]. The support provided by BPEL 1.1 and in some cases also by one of its implementations is discussed for the control-flow, data, resource, service interaction, and correlation patterns [2, 17–20, 23]. The process instantiation patterns and correlation patterns [1, 5] are evaluated for BPEL 2.0. Further pattern-based analyses of BPEL 1.1 and comparisons to other Web Services composition languages can be found in [6, 25]. For WF, only a single study that assesses its support for control-flow patterns in its revision 3.5 can be found so far [27]. All these studies use the traditional support measure or do not qualify the degree of support at all.

In this paper, we evaluate the complexity of a process model through the computation of a distance metric. Other attempts for measuring this complexity focus on the structure of the execution sequences produced by a process [4] or count the number of *and*, *or*, and *xor* nodes in its graph [21]. The *edit distance*, or *Levenshtein distance* [11], originally measures the distance between strings by counting the minimal amount of insertions, substitutions or deletions of characters that are needed to transform one string into the other. Here, we address process models instead of strings. The *graph edit distance* [7] has been put forward for this use case. It computes the distance between two process models using the amount of insertions, deletions and substitutions of nodes and control-flow edges in the process graph. Such edit distances have various applications in the area of service-oriented systems, for example in *service discovery* of composite services [26]. There, they are used to match a set of processes to a given query. Here however, our aim is to accurately classify the cost of changing process models. We argue that the graph edit distance is too abstract to be used for the problem at hand. It does not consider the configuration of nodes or edges and does not consider crucial characteristics that are typical for orchestration languages, such as variables or correlation sets. Consequently, its application would bear the same issues as the traditional support measure. Therefore, we specialize the set of underlying edit operations to capture the specifics of orchestration languages. A specialized set of edit operations allows for a more accurate cost estimation, than a mere comparison of nodes and edges would.

3 Edit Distance Support Measure

Our support measure is based on the idea to measure the degree of support provided by a language through the computation of the distance between an

[1] Please refer to `http://wiki.open-esb.java.net/Wiki.jsp?page=BPELSE`.

executable *process stub* without specific functionality and a process implementing a given pattern [10]. A process stub is a minimal process definition, a process model that forms the typical starting point of any realistic process. This process stub can be extended with the language constructs that are needed to implement exactly a single pattern. The distance between these two process models, the process stub and the pattern implementation, computed in a metrical scaling, provides a notion of complexity for the implementation of a pattern in a language. It is of importance that the process stub and the computation of the distance metric are similar for different languages. Also, the languages have to reside on a similar level of abstraction, which is the case for the languages in focus here. Omitting this requirement would not allow for meaningful results when directly comparing different languages.

Similarity between the process stubs of different languages can not be based on syntactical similarity because the languages have a different syntax. However, the process stubs can be implemented according to the same abstract scheme which will be presented in the next section, along with code samples[2] for the process stubs in BPEL 2.0, Sun BPEL, and WF 4. In the same fashion, the computation of the distance between the process stub and a pattern implementation using an edit distance, must be based on the same set of edit operations, even for different languages. Therefore, we go on with defining an abstract set of edit operations for orchestration languages and describe the mapping of these edit operations to the specifics of the languages in focus.

3.1 Common Schema for Process Stubs

To obtain distance values that are comparable for multiple languages, it is necessary to have a common basis, i.e., to use a semantically equivalent process stub. In this study, we examine multiple orchestration languages. Therefore, also the process stub ought to be applicable to orchestration languages in general. As a minimal feature of the executability of an orchestration model, it should provide the ability to create new process instances. In an orchestration, this is typically done using a *single event trigger* [5] such as an incoming message. Activities that process incoming messages, often called *receive* activities, are most convenient for this purpose. The main aim of the process stub is to extend it with the implementation of pattern. This can be achieved with a *sequence* activity. For the reason of extensibility, it is beneficial to be able to direct some input to a process instance. The initial message that triggers the creation of a new process instance is sufficient for this. Normally, this requires the definition of a variable in the process model, and a mapping from the input message to this variable. In BPEL[3] there are `receive` and `sequence` activities for these purposes.

[2] All code samples in this paper are limited to the most crucial aspects that are needed for understanding. Certain features, such as XML namespaces, are omitted completely.

[3] For the sake of brevity, we will not discuss the specifics of Sun BPEL separately from BPEL. Naturally, the languages are similar to a large extent. In cases where differences exist, we will make this explicitly clear.

To have input data available for a process instance, a `variable` definition and its reference in the `receive` activity are needed. Finally, also the WSDL interface for the BPEL process needs to be imported and used in a `myRole partnerLink`. List. 1 (taken from [10]) outlines such a process stub in BPEL 2.0.

Listing 1. Process stub for BPEL 2.0 and Sun BPEL

```
<process>
   <import location="ProcessInterface.wsdl" />
   <partnerLinks>
       <partnerLink name="MyPartnerLink" myRole="patternRole" />
   </partnerLinks>
   <variables>
       <variable name="StartProcessInput" messageType="int"/>
   </variables>
   <sequence>
        <receive createInstance="yes" variable="StartProcessInput"
         partnerLink="MyPartnerLink" operation="StartProcess"/>
         <!--Pattern Implementation-->
   </sequence>
</process>
```

In WF, the process stub looks very similar. An orchestration can be implemented as a *workflow service* and `Receive` and `Sequence` activities with the same semantics as in BPEL are available. `Variables` are scoped differently in WF and the definition of the service interface is not needed, but is inferred by the workflow engine from the messaging activities in the workflow. List. 2 outlines the process stub for WF.

Listing 2. Process stub for WF 4

```
<WorkflowService>
    <Sequence>
     <Sequence.Variables>
            <Variable Name="InstanceID" TypeArguments="Int32"/>
     </Sequence.Variables>
     <Receive CanCreateInstance="True" OperationName="StartProcess">
            <ReceiveParametersContent>
             <OutArgument Key="InputData">[InstanceID]</OutArgument>
            </ReceiveParametersContent>
     </Receive>
     <!--Pattern Implementation-->
    </Sequence>
</WorkflowService>
```

These process stubs are now enriched with the implementation of a particular pattern. The distance between a process stub and the pattern implementation is used to determine the degree of support provided for a pattern by the languages. It reflects the effort needed by the implementer of a pattern and the complexity of the implementation.

3.2 Edit Operations

The idea for calculating the edit distance here is to count all meaningful *implementation steps* that are needed by a user to achieve the functionality required by a pattern definition. Therefore, all relevant changes in a process model should

be covered and not just changes to nodes and edges in the process graph. In the following, we present a set of edit operations for orchestration languages and describe how they are represented in BPEL 2.0 and WF 4. The operations represent self-contained semantical units that stand for atomic implementation steps from a programmer perspective. As an example, adding a variable to a process model and setting its name and type, is considered as a single implementation step. These steps are independent of the serialization format or graphical representation of a language. This means that, although they might require several changes to the code of a process model or several actions in a graphical editor, they capture a single semantic change of a feature of the process model. Changes to process models can be facilitated by using a sophisticated development environment when implementing a pattern. But in taking into account features such as auto-completion or the macros of an editor, an analysis would no longer evaluate a language, but rather an environment for the language. As this is not our intention, we abstract from any tooling available when calculating the distance values.

The set of edit operations has been identified as follows: Starting with a list of candidate operations, the relevance of an operation was determined by the fact that it was needed to implement the behaviour required by a pattern. During the implementation of certain patterns it became obvious that further edit operations had to be defined, especially for messaging activities, and the list was extended. This procedure is similar to the method used by several authors for extrapolating a set of patterns from a set of real-world process models or the capabilities of several workflow systems [2, 8, 17, 18, 22–24]. No other edit operations apart from the operations described were necessary and it was possible, using this set of operations, to calculate selective and meaningful support values (cf. Sect. 4). More than 150 process models[4] were developed and these models serve as empirical evidence for the relevance of the edit operations from a programmer perspective. Each of the following operations adds one point to the edit distance of a solution to a pattern.

Insert Activity: *The insertion or substitution of an activity and the setting of the activity name.*
Any BPEL 2.0 activity and any WF 4 activity is covered by this operation. Further configuration of an inserted activity is not included. In a block-structured process model, activities are necessarily nested. There, inserting an activity also includes the modification of a composite activity (e.g. inserting a child activity). Therefore, the modification of the composite activity is also included in the insertion. For example, inserting an activity into an `onMessage` activity in BPEL 2.0 also modifies it.

Insert Edge: *The insertion of a control-flow edge and the setting of its name.*
This operation is generally only available in a graph-oriented model. In BPEL

[4] As discussed, we considered a total of 70 patterns. Because we analyzed three languages, there are up to three process models per pattern. Moreover, several patterns, especially the service interaction patterns [2], require more than one process model for a valid implementation.

2.0, edges are represented by `links` in a `flow` activity. In WF 4, edges are represented by `FlowSteps` in a `Flowchart` activity. An insertion of an edge in a graph also includes the setting of its target and source. All further configuration, such as the setting of a condition for the activation of an edge is not included in its insertion. For all other consideration, edges can be treated just as activities.

Insert Auxiliary Construct: *The insertion of a process element, apart from nodes and edges.*
Apart from activities and edges, languages may use a variety of auxiliary constructs that can be defined in a process model and be used by the activities of a process. Such elements are, for example, variables, correlations, or references to partners involved in the process. The insertion of such an element involves its initial configuration, such as the setting of its name and type. In BPEL 2.0, such constructs are `variables`, `correlationSets`, and `partnerLinks`. For a `variable`, the name and type must be fixed. For a `correlationSet`, the name and properties must be specified and for a `partnerLink`, the `name`, `partnerLinkType`, and `role` must be declared. In WF 4, the only additional process elements are `Variables`. Correlations can be defined using variables of a specific type, `CorrelationHandle`, and references to partners are not defined explicitly, but are contained in the configuration of messaging activities. In WF 4, the insertion of a `Variable` involves the setting of its type, name, and optionally a default value.

Configure Messaging Properties: *The setting of the messaging properties in a messaging activity.*
Messaging activities require the configuration of several properties, all related to the interface of the service to which they correspond. Typically, this is the setting of a service name and an operation name. The operation name marks an operation provided by the service which is identified by the service name. As they are logically related, these configurations are captured in a single edit operation. In BPEL 2.0, the configuration of the messaging properties of an activity involves the setting of the `partnerLink`, `portType` and `operation`. In WF 4, it corresponds to the setting of the `ServiceContractName` and the `OperationName`.

Configure Addresses: *The setting of an endpoint address.*
For an outbound messaging activity, apart from the messaging properties that relate to the interface of the service, also the address of a concrete service instance needs to be set. Otherwise, a messaging activity would not be able to direct a message to it. In BPEL 2.0, this is not necessarily needed, as the address of a service may be inferred from the `partnerLink` used in the activity which needs to be set as part of the messaging properties. This is not the case for WF 4, as it does not make use of an explicit predefined specification of the partners a process interacts with. Here, the relevant addresses need to be set separately for each messaging activity. Setting addresses can be done in several ways, such as via an `Endpoint` element and the setting of its `AddressUri` and the `Binding` to be used.

Configure Correlations: *The configuration of message correlation.*
In general, the configuration of message correlation requires a mapping from the parameters available to an activity to a predefined correlation variable. In BPEL 2.0, this operation involves the definition of a `correlations` and a `correlation` element, the setting of its name, and potentially whether the correlation set should be initiated. In WF 4, a `CorrelationHandle` needs to be referenced and a query that determines the elements of the parameters of the activity that identify the correlation needs to be specified.

Configure Parameters: *The setting of the input or output parameters of messaging activities.*
In BPEL 2.0, this implies referencing a predefined `variable`. If the parameters capture an outbound message, concrete data has to be assigned to this variable in a separate activity in advance to its use as a messaging parameter. The insertion of the activity that performs this assignment and its configuration is *not* covered by this edit operation. In WF 4 it implies the definition of a parameter and the mapping of the parameter to a predefined `Variable` using an expression.

Configure Expression: *The setting of an attribute value of an activity, edge, or auxiliary construct to an expression.*
Several attributes of process elements may require expressions as values. Examples are logical expressions used in the termination condition of a looping activity. The construction of such an expression may require several steps and may involve the use of several operators in a dedicated expression language (which is XPath 1.0 for BPEL 2.0 and Visual Basic for WF 4). The construction of an expression is rated as a single edit operation.

Assignment: *The setting of the content of an assign activity.*
The assignment of a value to a variable involves the specification of the variable name and the expression that produces the value which is assigned. While WF 4 allows for single assignments in an `Assign` activity only, BPEL 2.0 allows for multiple ones using multiple `copy` elements.

Change Configuration: *The change of any default value of an activity, edge or auxiliary construct to another value.*
All other changes of the configuration of the elements of a process can be represented by this operation. In BPEL 2.0 or WF 4, this could for example be the change of an attribute value or the setting of a child element of an activity. Several activities capture their configuration in child elements. An example is an `onAlarm` in BPEL 2.0 where the configuration of the wait condition is fixed in a `for` or `until` element.

The goal of the edit operations chosen is to measure the effort of implementing a pattern from a programmer perspective. The granularity of this set of operations essentially is a design choice and it would be an option to decrease or increase it. For instance, rating the insertion of an activity and its complete configuration as a single operation decreases the granularity, while counting each modification of an attribute of a construct increases it. Although we did not perform a complete analysis, we calculated the distance values for a subset of the process models with

these two modifications and observed the effect this had on the selectivity of the measure and its ability to characterize the complexity of the process models. It turned out that in neither case, the ability of the measure to discriminate between the languages (cf. Sect. 4) differed strongly. The problem with the first approach is that it assigns similar distance values to the implementations of control-flow and service interaction patterns, in spite of the fact that the latter describe more complicated scenarios. This is the case, because even complex messaging activities are as costly as more simple activities. As an example, a sequence of two `assign` activities would have the same distance value as a sequence of a `send` and a `receive` activity, although the latter two are more complicated to configure. In the second approach, activities that are rather simple but needed frequently and require extensive configuration tend to hide the complexity of other activities where the setting of attribute values corresponds to important design decisions. Examples are `assign` activities which require the specification of sources, targets, and possibly expressions or type specifications, and easily outweigh the configuration of a looping activity to execute in parallel. To sum up, the set of edit operations presented here provides, in our point of view, an acceptable trade off between the selectivity of the measure and its ability to assess the complexity of a pattern. An empirical study is needed to assess whether the complexity values calculated with the measure really correspond to the impression of human implementers.

From a theoretical point of view, edit distances that do not use the same weights for insertion and deletion operations are *quasi-metrics* [29, p. 12], as they do not satisfy the property of symmetry. Here, we assign higher costs to insertions, as we specify a number of fine-grained insertion operations. Deletions can only be achieved indirectly through substitution. The upper bound of the computation complexity of the distance value of a process model to a process stub is linear to the number of activities and auxiliary constructs used, multiplied with the maximum number of configuration options available for an activity.

3.3 Calculation Example

Next, we describe an implementation of the *Racing Incoming Messages* pattern [2] and present a code sample for the executable process in WF 4. This pattern describes a scenario where a party awaits one out of a set of messages. The messages may be of different structure, originate from different parties and be processed in a different manner, depending on their type. This aspect is well understood in all of the languages in focus here, although the solutions differ slightly. The implementation presented is motivated by the solution to the pattern in BPEL 1.1 described in [2].

The implementation in WF 4 builds upon the `Pick` activity. This activity contains multiple `PickBranch` activities, one for each of the messages that can be received. A minimal realization of the pattern contains two alternative messages. Each `PickBranch` activity contains a `Trigger`. Any WF activity can serve as `Trigger` and as soon as the `Trigger` completes, the according body is executed. When a `Trigger` completes, all other `PickBranches` are canceled.

Listing 3. Racing Incoming Messages pattern in WF

```
<WorkflowService>
 <Sequence>
  <Pick>
   <PickBranch>
    <PickBranch.Trigger>
     <Receive ServiceContractName="RacingIncomingMessages"
          OperationName="ReceiveMessageA" CanCreateInstance="True"/>
    </PickBranch.Trigger>
    <!-- Process MessageA -->
   </PickBranch>
   <PickBranch>
    <PickBranch.Trigger>
     <Receive ServiceContractName="RacingIncomingMessages"
          OperationName="ReceiveMessageB" CanCreateInstance="True"/>
    </PickBranch.Trigger>
    <!-- Process MessageB -->
   </PickBranch>
  </Pick>
 </Sequence>
</WorkflowService>
```

This structure is outlined in List. 3. The following edit operations add to the edit distance of this process compared to the process stub presented in List. 2:

1. **Insert Activity**: Substitute the `Receive` activity from the process stub with the `Pick` activity.
2. **Insert Activity**: Insert the first `PickBranch` activity.
3. **Insert Activity**: Insert the first `Receive` activity into the `Trigger` of the first `PickBranch` activity.
4. **Configure Messaging Properties**: To be able to receive messages, the `OperationName` and `ServiceContractName` of the first `Receive` activity has to be set.
5. **Configure Activity**: The `Receive` activity must be able to create a new process instance (by setting its `CanCreateInstance` attribute to `true`), otherwise the executable process would not be valid.
6. **Insert Activity**: Insert the second `PickBranch` activity.
7. **Insert Activity**: Insert the second `Receive` activity into the `Trigger` of the second `PickBranch` activity.
8. **Configure Messaging Properties**: To be able to receive messages, the `OperationName` and `ServiceContractName` of the second `Receive` activity has to be set.
9. **Configure Activity**: Set the `CanCreateInstance` attribute of the second `Receive` activity to `true`.

In total, this adds up to an edit distance of nine. There are similar process models in BPEL 2.0 and Sun BPEL, based on the `pick` activity. BPEL 2.0 scores an edit distance of eight, and Sun BPEL of ten (cf. Table 2). The edit distance allows to see that there are subtle differences in the degree of support provided by the languages. Things are different, when using the traditional trivalent measure. There, the solution achieves direct support, as there is a single essential activity (the `Pick`) implementing the pattern. This valuation could be questioned, as there is undoubtedly more than one activity involved in the solution. Nevertheless, this

valuation corresponds to the assumptions made in other evaluations [2, 6]. The same applies to the solutions in BPEL 2.0 and Sun BPEL. They also result in a rating of direct support. So when using the traditional measure, this pattern reveals no difference in the support provided by WF 4 , BPEL 2.0, or Sun BPEL.

4 Results and Evaluation

In the following sections, we present the results of an assessment of the languages WF 4, BPEL 2.0, and Sun BPEL for the *control-flow patterns* [19,23], the *service interaction patterns* [2], the *time patterns* [8] and the *patterns for changes in predefined regions* [24]. A detailed discussion of all these patterns and the solutions to them in the respective languages is not possible in the context of this paper. Therefore, the following sections present the overall results of the analysis and discuss their implications. We refer the interested reader to a technical report [9] for a description of every pattern and a discussion of the solutions in each of the languages. All process models that have been developed are available.[5] Our intention when developing the process models was to minimize the edit distance while providing a valid solution to a pattern. More efficient solutions to the patterns in terms of computing complexity may be possible.

4.1 Control-flow Patterns

Table 1 outlines the results for the control-flow patterns [19, 23] and compares them to the results of studies which analyzed preceding versions of BPEL and WF. The control-flow patterns describe typical structures of the control-flow perspective of automated processes. Our solutions to the patterns in the newer language revisions were motivated by preceding studies [19, 25, 27], given required language constructs were still in place in the newer language versions. The results of preceding studies only present the trivalent support measure.

Table 1 reveals that the edit distance support measure provides a higher degree of selectivity among the languages than the traditional trivalent measure does. The number of solutions in both, BPEL 2.0 and WF 4, can be used to assess the quality of the languages, but also to quantify the degree of selectivity provided by a support measure. This quantification is based on the number of solutions where a support measure discriminates between the languages in relation to the total number of solutions to the same patterns. A value of 1 for this relation states that a support measure completely discriminates in all cases. A value of 0 states that a support measure discriminates in no case. For 30 of the 43 patterns here, solutions could be found in WF 4. In BPEL 2.0, 31 patterns could be implemented. For 29 patterns, solutions could be found in both, WF 4 and BPEL 2.0. Only in six of these cases, the trivalent measure discriminates, so the degree of selectivity amounts to $6/29 = 0.21$. The edit distance discriminates

[5] The process models can be downloaded at `http://www.uni-bamberg.de/pi/orch-pattern`. [9] also contains a description on how to execute them.

Table 1. Support of workflow control-flow patterns. If available, the edit distance is displayed first followed by the trivalent measure in parentheses. A value of '-' for the edit distance means that no valid solution could be found. As opposed to this, a value of '-' for the trivalent measure means that either no valid solution could be found or that all possible valid solutions require the use of more than two constructs.

Pattern	**WF 3.5** taken from [27]	**WF 4**	**BPEL 1.1** taken from [19]	**BPEL 2.0**	**Sun BPEL**
Basic Patterns					
WCP-1. Sequence	+	2 (+)	+	2 (+)	2 (+)
WCP-2. Parallel Split	+	3 (+)	+	3 (+)	3 (+)
WCP-3. Synchronization	+	3 (+)	+	3 (+)	3 (+)
WCP-4. Exclusive Choice	+	4 (+)	+	4 (+)	4 (+)
WCP-5. Simple Merge	+	4 (+)	+	4 (+)	4 (+)
Advanced Branching and Synchronization Patterns					
WCP-6. Multi-Choice	+	7 (+/-)	+	7 (+/-)	7 (+/-)
WCP-7. Structured Synchronizing Merge	+	7 (+/-)	+	7 (+/-)	7 (+/-)
WCP-8. Multi-Merge	-	- (-)	-	- (-)	- (-)
WCP-9. Structured Discriminator	+/-	9 (+/-)	-	10 (-)	10 (-)
WCP-28. Blocking Discriminator	-	- (-)	-	- (-)	- (-)
WCP-29. Cancelling Discriminator	+	9 (+)	-	10 (-)	10 (-)
WCP-30. Structured Partial Join	+/-	12 (+/-)	-	31 (-)	31 (-)
WCP-31. Blocking Partial Join	-	- (-)	-	- (-)	- (-)
WCP-32. Cancelling Partial Join	+	12 (+)	-	31 (-)	31 (-)
WCP-33. Generalized AND-Join	-	- (-)	-	- (-)	- (-)
WCP-37. Acyclic Synchronizing Merge	+/-	- (-)	+	11 (+)	- (-)
WCP-38. General Synchronizing Merge	-	- (-)	-	- (-)	- (-)
WCP-41. Thread Merge	-	- (-)	+/-	- (-)	- (-)
WCP-42. Thread Split	-	- (-)	+/-	- (-)	- (-)
Multiple Instances (MI) Patterns					
WCP-12. MI without Synchronization	+	6 (+)	+	7 (+)	12 (+/-)
WCP-13. MI with a priori Design-Time Knowledge	+	6 (+)	+	7 (+)	19 (+/-)
WCP-14. MI with a priori Run-Time Knowledge	+	6 (+)	-	7 (+)	- (-)
WCP-15. MI without a priori Run-Time Knowledge	-	- (-)	-	- (-)	- (-)
WCP-34. Static Partial Join for MI	+/-	10 (+/-)	-	8 (+/-)	- (-)
WCP-35. Cancelling Partial Join for MI	+	10 (+)	-	8 (+)	- (-)
WCP-36. Dynamic Partial Join for Multiple Instances	-	- (-)	-	- (-)	- (-)
State-based Patterns					
WCP-16. Deferred Choice	+	9 (+/-)	+	8 (+)	10 (+)
WCP-17. Interleaved Parallel Routing	+	- (-)	+/-	16 (+/-)	- (-)
WCP-18. Milestone	+	11 (+/-)	-	11 (+/-)	11 (+/-)
WCP-39. Critical Section	+	9 (+/-)	+	15 (+/-)	40 (-)
WCP-40. Interleaved Routing	+	9 (+/-)	+	15 (+/-)	40 (-)
Cancellation Patterns					
WCP-19. Cancel Activity	+	9 (+/-)	+	8 (+/-)	8 (+/-)
WCP-20. Cancel Case	+	4 (+)	+	3 (+)	3 (+)
WCP-25. Cancel Region	+	9 (+/-)	+/-	8 (+/-)	8 (+/-)
WCP-26. Cancel MI Activity	+	14 (+/-)	-	13 (+/-)	55 (-)
WCP-27. Complete MI Activity	-	10 (+)	-	8 (+)	- (-)
Iteration Patterns					
WCP-10. Arbitrary Cycles	+	17 (-)	-	18 (-)	18 (-)
WCP-21. Structured Loop	+	5 (+)	+	5 (+)	5 (+)
WCP-22. Recursion	-	- (-)	-	- (-)	- (-)
Termination Patterns					
WCP-11. Implicit Termination	+	0 (+)	+	0 (+)	0 (+)
WCP-43. Explicit Termination	+	9 (+/-)	-	6 (+)	6 (+)
Trigger Patterns					
WCP-23. Transient Trigger	+	- (-)	-	- (-)	- (-)
WCP-24. Persistent Trigger	+	0 (+)	+	0 (+)	0 (+)

in 18 cases, so this number amounts to $18/29 = 0.62$. For all 25 patterns to which solutions could be found in Sun BPEL, also solutions in WF 4 could be found. Here, the degree of selectivity of the trivalent measure amounts to $11/25 = 0.44$, and for the edit distance it amounts to $14/25 = 0.56$.

It is not surprising that the degree of support for several patterns, such as *Parallel Split* or *Exclusive Choice*, is identical in WF 4 and BPEL 2.0 even using the edit distance. These patterns relate to concepts that are very common and consequently the solutions are very similar. For several patterns, such as the *Discriminator* and *Partial Join* patterns in BPEL 2.0, it is interesting to see that there is no support according to the trivalent measure, but the edit distance shows that they are relatively easy to implement.

The degree of pattern support has changed marginally from BPEL 1.1 to BPEL 2.0. There are few differences in the set of activities available and only the new parallel `forEach` activity has an impact on the support for control-flow patterns provided by the language. With the help of this activity, several of the *Multiple Instances* patterns can be supported. More efficient solutions to the *Discriminator* and *Partial Join* patterns in BPEL 2.0 could be implemented by providing a `completionCondition` for the `flow` activity similar to the `forEach` activity. BPEL 2.0 does not support several patterns due to its structuredness and the inability to create cycles using `links`, as well as its threading model. The support provided by Sun BPEL is severely limited by its lack of `links`, isolated `scopes` and parallel `forEach` activities.

When comparing WF 3.5 to WF 4, one thing becomes obvious: While the solutions of the patterns have changed considerably (cf. [9]), there is little change in the overall degree of support provided by the language. All in all however, fewer patterns are supported. There are two main reasons for this. First, the lack of the state machine modeling style that was present in WF 3.5 limits the support. This modeling style was especially suited to provide elegant solutions to state-based patterns and several other patterns for unstructured process models. In April 2011, Microsoft reacted to the demands of the community of WF users and re-introduced the state machine modeling style for WF 4 in its first platform update of .NET 4.[6] Second, while the new flowchart modeling style provides an excellent means for building unstructured, graph-oriented process models, it is not able to live up to its full potential, due to its inability to describe concurrent branches. A *Parallel Split* or *Multi-Choice* construct is yet missing in this style.

Altogether, the support provided by WF 4 and BPEL 2.0 is still very similar. Things are different when comparing Sun BPEL and WF 4. WF 4 provides support for more patterns and the solutions are often also less complex.

4.2 Service Interaction Patterns

Service interaction patterns [2] describe interaction scenarios that are typical in the B2B domain. Due to their distributed nature, nearly all service interaction patterns must be implemented by more than one process. In most cases there is an

[6] The documentation is available at `http://support.microsoft.com/kb/2478063`.

initiator process that starts a communication session and one or more responder processes. The edit distance that describes the support for a pattern is the sum of the edit distances of all the processes involved. The results of the analysis for the service interaction patterns are outlined in Table 2. The degree of support

Table 2. Support of Service Interaction Patterns

Pattern	WF 4	BPEL 2.0	Sun BPEL
Single-Transmission Bilateral Patterns			
SIP-1 Send	4 (+)	7 (+)	7 (+)
SIP-2 Receive	4 (+)	7 (+)	7 (+)
SIP-3 Send/Receive	9 (+)	15 (+/-)	15 (+/-)
Single-Transmission Multi-lateral Patterns			
SIP-4 Racing Incoming Messages	9 (+)	8 (+)	10 (+)
SIP-5 One-to-Many Send	9 (+)	12 (+)	12 (+)
SIP-6 One-from-Many Receive	37 (+/-)	49 (+/-)	49 (+/-)
SIP-7 One-to-Many Send/Receive	36 (+/-)	- (-)	- (-)
Multi-Transmission Bilateral			
SIP-8 Multi Responses	71 (-)	90 (-)	90 (-)
SIP-9 Contingent Requests	28 (+)	34 (+)	34 (+)
SIP-10 Atomic Multicast Notification	40 (-)	- (-)	- (-)
Routing Patterns			
SIP-11 Request with Referral	21 (+)	28 (+)	- (-)
SIP-12 Relayed Request	31 (+/-)	47 (+/-)	- (-)
SIP-13 Dynamic Routing	- (-)	- (-)	- (-)

for the service interaction patterns provided by WF 4 is considerably better than that of BPEL 2.0. WF 4 provides a wider range of messaging activities and its correlation mechanism is less restrictive. WF 4 supports more patterns and, as the edit distance demonstrates, almost all solutions are less complex. As before, Sun BPEL falls behind the other two languages in both, the number of patterns supported and the complexity of the solutions. Especially its lack of support for dynamic partner binding is critical, resulting from the inability to re-assign endpoint references to `partnerLinks`.

4.3 Time Patterns

Time patterns [8] mark typical time-related constraints of the control-flow perspective of processes. The results of the analysis of the support for time patterns are given in Table 3. The support for time patterns relies heavily on the representation for dates and times and the expression languages available. WF 4 uses sophisticated data types and time-based operations from the .NET class library. BPEL 2.0 requires only the support for XPath 1.0 as expression language, which completely lacks time-based operations. Sun BPEL incorporates some time-based functions of XPath 2.0 and thus allows to increase the degree of pattern support. In fact, for this pattern catalog, Sun BPEL excels BPEL 2.0. By consolidating the BPEL standard to also require the support for XPath 2.0 as expression language, BPEL would achieve a similar degree of support as WF 4.

Table 3. Support of Time Patterns

Pattern	WF 4	BPEL 2.0	Sun BPEL
Durations and Time Lags			
TP-1. Time Lags between two Activities	8 (+)	- (-)	- (-)
TP-2. Durations	6 (+/-)	7 (+/-)	7 (+/-)
TP-3. Time Lags between Events	8 (+)	- (-)	- (-)
Restrictions of Process Execution Points			
TP-4. Fixed Date Elements	3 (+)	3 (+)	3 (+)
TP-5. Schedule Restricted Elements	3 (+)	- (-)	- (-)
TP-6. Time Based Restrictions	6 (+)	- (-)	- (-)
TP-7. Validity Period	4 (+)	- (-)	4 (+)
Variability			
TP-8. Time Dependent Variability	3 (+)	11 (+/-)	4 (+)
Recurrent Process Elements			
TP-9. Cyclic Elements	12 (+/-)	- (-)	- (-)
TP-10. Periodicity	8 (+/-)	7 (-)	7 (-)

4.4 Patterns for Changes in Predefined Regions

Patterns for changes in predefined regions are a subset of the change patterns [24]. They describe structures that allow for changes in the control-flow perspective of processes at run-time. In most cases, these structures can be captured using certain control-flow patterns. As can be seen in Table 4, WF 4 and BPEL 2.0 are

Table 4. Support of Patterns for Changes in Predefined Regions

Patterns for Changes in Predefined Regions	WF 4	WS-BPEL 2.0	Sun BPEL
PP-1. Late Selection of Process Fragments	9 (+/-)	8 (+)	10 (+)
PP-2. Late Modeling of Process Fragments	- (-)	- (-)	- (-)
PP-3. Late Composition of Process Fragments	9 (+/-)	15 (+/-)	40 (-)
PP-4. Multiple Instance Activity	6 (+)	7 (+)	- (-)

roughly equivalent concerning their support for patterns for changes in predefined regions. On average, the solutions in WF 4 are less complex. In any case WF 4 and BPEL 2.0 excel Sun BPEL.

5 Conclusion and Outlook

This study introduced an edit distance-based measure for assessing pattern support that overcomes granularity and selectivity issues of the traditional measure. Its applicability was assessed by an analysis of the orchestration languages BPEL 2.0, Sun BPEL, and WF 4. The use of this support measure for calculating the degree of support overcomes the problems the traditional trivalent support measure posed on preceding analyses. What is more, it gives a notion for the complexity of a solution to a pattern in a language and the effort required by its implementer. Also, it is directly comparable across the boundaries of languages and pattern catalogs. Future analyses can provide more meaningful and selective results by relying on this edit distance support measure.

Furthermore, the results show that WF 4 excels both, BPEL 2.0 and its implementation Sun BPEL, concerning the degree of pattern support. BPEL 2.0 and WF 4 are largely equivalent concerning their degree of support for control-flow and change patterns. Things are different when looking at the service interaction and time patterns. WF 4 supports two service interaction patterns that are not supported by BPEL 2.0 and more than twice as many time patterns. Furthermore, for almost all time and service interaction patterns, the solutions are less complex in WF 4. For Sun BPEL, the analysis demonstrates that its degree of pattern support is rather limited.

Future work concentrates on the automation of the calculation of the edit distance. The edit operations presented here can serve as foundation for a unified model of orchestration languages that allows for an automated calculation of distance values. To fully automate the computation, it is necessary to construct a mapping from a concrete orchestration language to this model. Another open issue is the assessment of the efficiency and scalability of the solutions described here. As discussed, we cannot guarantee that we have found the most efficient solutions to all patterns in the languages in focus. A community approach, starting with the results from [9] and involving researchers from other institutions, might help to optimize the process models. Also the analysis of closely related languages, such as BPMN 2.0, is an interesting field of study.

References

1. A. P. Barros, G. Decker, M. Dumas, and F. Weber. Correlation Patterns in Service-Oriented Architectures. In M. B. Dwyer and A. Lopes, editors, *FASE*, volume 4422 of *LNCS*, pages 245–259, Braga, Portugal, March/April 2007. Springer, Heidelberg.
2. A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service Interaction Patterns. In *BPM*, pages 302–318, Nancy, France, September 2005.
3. B. Bukovics. *Pro WF: Windows Workflow in .NET 4*. Apress, June 2010. ISBN-13: 978-1-4302-2721-2.
4. J. Cardoso. Business Process Quality Metrics: Log-Based Complexity of Workflow Patterns. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *LNCS*, pages 427–434. Springer, Heidelberg, 2007.
5. G. Decker and J. Mendling. Process Instantiation. *Data and Knowledge Engineering, Elsevier*, 68:777–792, 2009.
6. G. Decker, H. Overdick, and J. Zaha. On the Suitability of WS-CDL for Choreography Modeling. In *EMISA*, pages 21–33, Hamburg, Germany, October 2006.
7. R. M. Dijkman, M. Dumas, and L. García-Bañuelos. Graph Matching Algorithms for Business Process Model Similarity Search. In *BPM*, pages 48–63, Ulm, Germany, September 2009.
8. A. Lanz, B. Weber, and M. Reichert. Workflow Time Patterns for Process-Aware Information Systems. In *BPMDS and EMMSAD in conjunction with CAiSE*, LNBIP, pages 94–107, Hammamet, Tunisia, June 2010. Springer, Heidelberg.
9. J. Lenhard. A Pattern-based Analysis of WS-BPEL and Windows Workflow. Technical Report 88, Otto-Friedrich-Universität Bamberg, March 2011. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik.

10. J. Lenhard, A. Schönberger, and G. Wirtz. Streamlining Pattern Support Assessment for Service Composition Languages. In *ZEUS*, volume 705 of *CEUR Workshop Proceedings*, pages 112–119, Karlsruhe, Germany, February 2011. CEUR-WS.org.
11. V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.
12. A. Norta, M. Hendrix, and P. Grefen. A Pattern-Knowledge Base Supported Establishment of Inter-organizational Business Processes. In R. Meersman, Z. Tari, and P. Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4277 of *LNCS*, pages 834–843. Springer, Heidelberg, 2006.
13. OASIS. *Web Services Business Process Execution Language*, April 2007. v2.0.
14. A. O'Hagan, S. Sadiq, and W. Sadiq. Evie - A developer toolkit for encoding service interaction patterns. *Information Systems Frontiers*, 11(3):211–225, 2009.
15. M. P. Papazoglou and D. Georgakopoulos. Service-oriented Computing. *Communications of the ACM*, 46(10):24–28, October 2003.
16. C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, October 2003.
17. N. Russell, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. F. e Cunha, editors, *CAiSE*, pages 216–232, Porto, Portugal, June 2005. Springer, Heidelberg.
18. N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *ER*, LNCS, pages 353–368, Klagenfurt, Austria, October 2005. Springer, Heidelberg.
19. N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical report, BPM Center Report, 2006.
20. N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In *CAiSE*, pages 288–302, Luxembourg, Luxembourg, June 2006. Springer.
21. L. Sánchez-González, F. Ruiz, F. García, and J. Cardoso. Towards Thresholds of Control Flow Complexity Measures for BPMN models. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1445–1450. ACM, 2011.
22. L. H. Thom, M. Reichert, and C. Iochpe. Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence. *IJBPIM*, 4(2):93–110, 2009.
23. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases, Springer*, 14(1):5–51, 2003.
24. B. Weber, S. Rinderle-Ma, and M. Reichert. Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. *Data and Knowledge Engineering, Elsevier*, 66(3):438–466, July 2008.
25. P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In *ER*, volume 2813 of *LNCS*, pages 200–215, Chicago, Illinois, USA, October 2003. Springer.
26. A. Wombacher and C. Li. Alternative Approaches for Workflow Similarity. In *IEEE SCC*, pages 337–345, Miami, Florida, USA, July 2010.
27. M. Zapletal, W. M. P. van der Aalst, N. Russell, P. Liegl, and H. Werthner. An Analysis of Windows Workflow's Control-Flow Expressiveness. In *ECOWS*, pages 200–209, Eindhoven, The Netherlands, November 2009.
28. U. Zdun and S. Dustdar. Model-driven and pattern-based integration of process-driven soa models. *IJBPIM*, 2(2):109–119, 2007.
29. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, Heidelberg, 2006. ISBN 978-0-387-29146-8.