BAMBERGER BEITRÄGE
ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK
ISSN 0937-3349

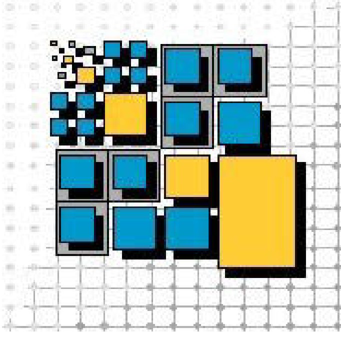
Nr. 88

A Pattern-based Analysis of WS-BPEL and Windows Workflow

Jörg Lenhard

March 2011

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK
OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG



Distributed Systems Group

Otto-Friedrich Universität Bamberg

Feldkirchenstr. 21, 96052 Bamberg, GERMANY

Prof. Dr. rer. nat. Guido Wirtz

`http://www.uni-bamberg.de/pi/`

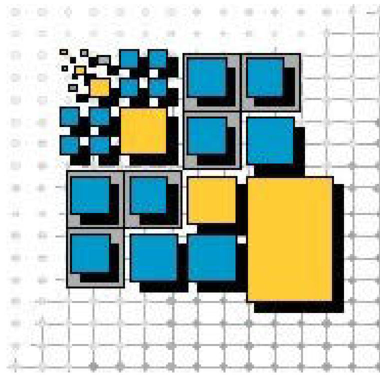
Due to hardware developments, strong application needs and the overwhelming influence of the net in almost all areas, distributed systems have become one of the most important topics for nowadays software industry. Owing to their ever increasing importance for everyday business, distributed systems have high requirements with respect to dependability, robustness and performance. Unfortunately, distribution adds its share to the problems of developing complex software systems. Heterogeneity in both, hardware and software, permanent changes, concurrency, distribution of components and the need for inter-operability between different systems complicate matters. Moreover, new technical aspects like resource management, load balancing and guaranteeing consistent operation in the presence of partial failures and deadlocks put an additional burden onto the developer.
The long-term common goal of our research efforts is the development, implementation and evaluation of methods helpful for the realization of robust and easy-to-use software for complex systems in general while putting a focus on the problems and issues regarding distributed systems on all levels. Our current research activities are focussed on different aspects centered around that theme:

- *Reliable and inter-operable Service-oriented Architectures*: Development of design methods, languages, tools and middle-ware to ease the development of SOAs with an emphasis on provable correct systems that allow for early design-evaluation due to rigorous development methods. Additionally, we work on approaches and standards to provide truly inter-operable platforms for SOAs.
- *Implementation of Business Processes and Business-to-Business-Integration (B2Bi)*: Starting from requirements for successful B2Bi development processes, languages and systems, we investigate the practicability and inter-operability of different approaches and platforms for the design and implementation of business processes with a focus on combining processes from different business partners.
- *Quality-of-Service (QoS) Aspects for SOA and B2Bi*: QoS aspects, especially reliability and security, are indispensable when putting distributed systems into practical use. We work on methods that allow for a seamless observance of QoS aspects during the entire development process from high-level business processes down to implementation platforms.
- *Agent and Multi-Agent (MAS) Technology:* Development of new approaches to use Multi-Agent-Systems for designing, organizing and optimizing complex systems ranging from service management and SOA to electronic markets and virtual enterprises.
- *Visual Programming- and Design-Languages*: The goal of this long-term effort is the utilization of visual metaphors and languages as well as visualization techniques to make design- and programming languages more understandable and, hence, more easy-to-use.

More information about our work, i.e., projects, papers and software, is available at our homepage (see above). If you have any questions or suggestions regarding this report or our work in general, don't hesitate to contact me at `guido.wirtz@uni-bamberg.de`

Guido Wirtz

Bamberg, January 2010



A Pattern-based Analysis of WS-BPEL and Windows Workflow

Jörg Lenhard

Lehrstuhl für Praktische Informatik, Fakultät WIAI

Abstract Orchestration languages are of paramount importance for building composite services in service-oriented architectures. Pattern-based analysis is a method that allows to determine the expressiveness of existing process languages and serves as a means of comparison between different languages. The aim of this study is the analysis and comparison of important languages for building Web Services-based orchestrations, as well as the improvement of the method of pattern-based analysis.
The predominant orchestration language today is the Web Services Business Process Execution Language (WS-BPEL) 2.0. This language is a standard that has been implemented by several companies and projects, such as the OpenESB BPEL Service Engine. An additional language is Windows Workflow 4 that is shipped by Microsoft as part of the .NET framework.
There are various aspects, represented by pattern catalogs, for which existing languages can be analyzed. This study suggests a methodology for ordering existing pattern catalogs according to their importance for a selected problem domain which is Business-to-Business Integration. It furthermore presents an extensive evaluation of the languages at hand and assesses the degree of support they provide for several of the most important pattern catalogs. These catalogs are the workflow control-flow patterns, the service interaction patterns, the change patterns and the time patterns.

Keywords Service-oriented Architecture, Composite Service, Orchestration Language, WS-BPEL, Windows Workflow, Pattern, Support Measure, Edit Distance

Contents

1 Assessing Service Composition Languages 1

2 Orchestration Languages 4
2.1 Fundamentals . . . 4
2.1.1 Business-to-Business Integration Schema . . . 4
2.1.2 Process Structure and Service-oriented Processes . . . 5
2.1.3 Web Services Technology . . . 7
2.2 Web Services Business Process Execution Language 2.0 . . . 8
2.2.1 Basic Concepts of WS-BPEL . . . 8
2.2.2 WS-BPEL Activities . . . 9
2.3 Windows Workflow 4 . . . 13
2.3.1 Basic Concepts of WF . . . 13
2.3.2 Base Activity Library . . . 16

3 Patterns for Orchestration Languages 21
3.1 Survey of Pattern Catalogs . . . 21
3.1.1 Workflow Patterns . . . 22
3.1.2 Service Interaction Patterns . . . 24
3.1.3 Integration Patterns . . . 25
3.2 Pattern Catalog Selection . . . 26
3.2.1 Approach . . . 26
3.2.2 Evaluation and Results . . . 27
3.3 Current State of Pattern-based Analysis . . . 32

4 Streamlining Pattern-based Analysis 33
4.1 Implementation Validity . . . 33
4.2 Edit Distance . . . 36

4.3 Trivalent Measure . . . 39

5 Analysis of Pattern Support **42**

5.1 Language Scope and Logging . . . 42

5.1.1 WF Scope and Environment . . . 43

5.1.2 WS-BPEL Scope and Environment . . . 44

5.2 Example Calculation . . . 45

5.3 Control-flow Patterns . . . 50

5.3.1 Analysis of Control-flow Patterns . . . 50

5.3.2 Discussion of Support for Control-flow Patterns . . . 76

5.4 Change Patterns . . . 79

5.4.1 Analysis of Patterns for Changes in Predefined Regions . . . 79

5.4.2 Discussion of Support for Patterns for Changes in Predefined Regions . . 81

5.5 Service Interaction Patterns . . . 82

5.5.1 Analysis of Service Interaction Patterns . . . 82

5.5.2 Discussion of Support for Service Interaction Patterns . . . 99

5.6 Time Patterns . . . 100

5.6.1 Analysis of Time Patterns . . . 102

5.6.2 Discussion of Support for Time Patterns . . . 110

5.7 Summary . . . 111

6 Conclusion **113**

Bibliography **114**

A Overview of Support Measures **118**

B Manual for WF Processes **119**

B.1 Environment Installation . . . 119

B.2 Folder and Project Structure . . . 120

B.3 Deploying and Executing Workflow Services 121
B.4 Logging in Workflows 122
B.5 Viewing Workflow Output 123

C Manual for WS-BPEL Processes **124**

C.1 Environment Installation 124
C.2 Folder and Project Structure 124
C.3 Related WSDL and XSD Files 125
C.4 Deployment Procedure 126
C.5 Executing Processes 127
C.6 Viewing Process Output 127

D List of previous University of Bamberg reports **128**

List of Figures

1 Business-to-Business Integration (B2Bi) Schema as defined in [41] 5
2 Workflow Structure 6
3 Overview of the Windows Workflow Foundation Architecture 14
4 Flowchart Workflow 15
5 Decision Tree for Determining Implementation Validity 35
6 Model of Arbitrary Cycle 73
7 Analysis Log in the Event Viewer 123

List of Tables

1 Classification of Pattern Catalogs 22

2 Ranking of Pattern Catalogs 31

3 Current State of Evaluation of WS-BPEL and Windows Workflow 32

4 Computation of the Trivalent Support Measure 41

5 Support of Workflow Control-flow Patterns 77

6 Degree of Selectivity of the Support Measures for the Control-flow Patterns . . . 78

7 Support of Patterns for Changes in Predefined Regions 81

8 Degree of Selectivity of the Support Measures for the Service Interaction Patterns 99

9 Support of Service Interaction Patterns 100

10 Support of Time Patterns 111

List of Listings

1 Exemplary Structure of a WS-BPEL Process 8

2 Process Stubs 35

3 Exemplary Use of a `LogCodeActivity` 44

4 Sun BPEL Trace Extension 45

5 Calculation Example for WF 46

6 Calculation Example for WS-BPEL 47

7 Sequence Pattern 50

8 Parallel Split and Synchronization Pattern 51

9 Exclusive Choice and Simple Merge Pattern 52

10 Multi-Choice and Structured Synchronizing Merge Patterns 52

11 Acyclic Synchronizing Patterns 55

12 Discriminator Patterns 57

13 Partial Join Patterns in WS-BPEL 58

14 Multiple Instances Patterns 61

15 Cancel Case Pattern 62

16 Cancellation Patterns in WF and WS-BPEL 64

17 Cancel Multiple Instance Activity Pattern in Sun BPEL 64

18 Complete Multiple Instance Activity Pattern 65

19 Deferred Choice Pattern 67

20 Interleaved Parallel Routing Pattern in WS-BPEL 68

21 Milestone Pattern 69

22 Critical Section Pattern 71

23 Structured Loop Pattern 72

24 Arbitrary Cycles Pattern 74

25 Send Pattern 83

26 Receive Pattern . . . 83
27 Send/Receive Pattern . . . 84
28 One-To-Many Send Pattern . . . 86
29 One-From-Many Receive Pattern . . . 87
30 One-To-Many Send/Receive Pattern in WF . . . 89
31 Multi-Responses Pattern in WF . . . 90
32 Multi-Responses Pattern in WS-BPEL . . . 91
33 Contingent Requests Pattern . . . 93
34 Atomic Multicast Notification Pattern in WF . . . 94
35 Request with Referral Pattern . . . 96
36 Relayed Request Pattern . . . 98
37 Maximum Time Lags between two Activities in WF . . . 103
38 Durations Pattern . . . 103
39 Fixed Date Element Pattern . . . 105
40 Schedule Restricted Element Pattern in WF . . . 106
41 Time Based Restrictions Pattern in WF . . . 106
42 Validity Period Pattern in Sun BPEL . . . 107
43 Time Dependent Variability in WS-BPEL . . . 108
44 Cyclic Elements Pattern in WF . . . 108
45 Periodicity Pattern . . . 109
46 Project Structure for WF Processes . . . 120
47 WriteLine Method of the LogWriter Class . . . 122
48 Project Structure for BPEL Processes . . . 124
49 Logging Output for WS-BPEL Processes . . . 127

List of Acronyms

API	Application Programming Interface
BAL	Base Activity Library
B2B	Business-to-Business
B2Bi	Business-to-Business Integration
BPEL	see WS-BPEL
CIL	Common Intermediate Language
CLR	Common Language Runtime
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
OASIS	Organization for the Advancement of Structured Information Standards
PAIS	Process-Aware Information System
SOA	Service-Oriented Architecture
SOAP	formerly known as Simple Object Access Protocol
SMTP	Simple Mail Transfer Protocol
QoS	Quality of Service
WCF	Windows Communication Foundation
WF	Windows Workflow
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Services Description Language
W3C	World Wide Web Consortium
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language
XSD	XML Schema Definition
XPath	XML Path Language
XQuery	XML Query Language

1 Assessing Service Composition Languages

Service-Oriented Architectures (SOAs) describe an architectural paradigm for building complex distributed systems based on services and have gained noticeable interest in recent years. Services are interface-based computing facilities which are described in a uniform, technology-neutral manner. They allow loosely-coupled, message-based interaction and are transparent concerning their location [7,32]. An infrastructure that provides basic management functionality for services, such as publishing, discovering, selecting and binding services forms the basic layer of a SOA [32]. This layer already offers a variety of advantages over preceding approaches ([16], pp. 70–80): First, the focus on a technology-neutral description of services leads to a higher level of interoperability of systems and a higher degree of independence from single software vendors. Second, in combination with the principle of loose coupling, this also allows for a higher degree of software reuse and quicker adaptation of software components to changing needs. Third, services can be used to encapsulate atomic units of business functionality. This resembles the notion of service in the business domain and helps to unify the technical and the business view in an enterprise.

Another powerful property of the SOA paradigm is the layer that is built on top of the basic one, the *service composition layer* [32]. This layer concerns the construction of composite services based on other services, which is often achieved by combining the calls to existing services in a process-like manner. A process generally consists of the control- and data-flow definition between the execution of different tasks. Using services instead of tasks, a composite service essentially defines the control- and data-flow dependencies between different service invocations [7]. Languages for describing composite services are called *service composition languages*. The explicit realization of processes on the basis of services can be used for a straight-forward mapping of the real-world processes in an enterprise to a technical realization. Given changes occur to the structure of the real-world processes, it is relatively easy to adapt to these changes by simply changing the control-flow of a composite service and without touching the implementations of the single services.

An important field of application for such service-based processes is *B2Bi* [7]. Here, several partners agree on a global process model that describes the interaction between several autonomous partners for a common business process. Subsequently, the global process model is partitioned into several local processes, one for each of the different partners [41]. In the context of SOAs, a model that describes a process from a global point of view is called a *choreography* model [7]. Here, autonomous partners interact without a central coordinator. If a process is described from the viewpoint of a central coordinator, for example a single company, involving only its local view and interactions it is called an *orchestration* [33]. With *Web Services* technology, a variety of standards and tools is at hand today for implementing SOAs. There are also several service composition languages for describing Web Services-based choreographies and orchestrations. In the case of orchestrations, the language most widely used today is the *Web Services Business Process Execution Language (WS-BPEL)* [31]. Recently, other languages, like *Windows Workflow (WF)* [5], have emerged.

Currently, academic debate centers around the question, which of these languages is most suitable for which scenarios, what features, and aspects such languages should incorporate

(only to mention some studies on this problem: [8,9,11,30,52,53,55]). Traditional notions such as Turing-completeness are inappropriate for capturing the suitability of service composition languages. This is because the purpose of such languages is not to solve arbitrary computable problems, but to describe processes based on services. That is why the quality of a language in this context is determined by its *domain appropriateness* [30]. This concept relates to the structures and constructs that are expressible in the language, also described as its *expressive power* [43]. A language is appropriate for a domain if it neither contains *construct deficits*, nor *construct excesses* [30]. This means that the language contains exactly the amount of constructs needed in a certain domain, neither too few (construct deficit), nor too many (construct excess). Put in other words, a language is appropriate for a certain domain if it provides an adequate amount of expressive power. A common method for determining the expressive power of process languages is *pattern-based analysis*. First introduced for workflow languages [43], this method evaluates a language for its support for certain sets of patterns that describe relevant language structures in an abstract form. Different languages can then be compared concerning their degree of support for these patterns. Consequently, a language that provides a higher degree of support is also more expressive than another language. If the sets of patterns are selected based on their importance to a domain, a higher expressiveness corresponds to a higher domain appropriateness.

Today, many pattern catalogs are available that could be used for pattern-based analysis of service composition languages [2, 3, 10, 19, 20, 23, 28, 35, 37–39, 42–44, 51]. However, a study that analyzes a language using multiple pattern catalogs faces several problems. The different publications use varying notions of what criteria must be fulfilled to achieve support for a pattern. This limits the comparability of the results for different pattern catalogs. In fact, most authors use different notions of what counts as support and also do not document clearly what criteria must be fulfilled by a candidate solution to offer support for a pattern. This way, the degree of support determined in an analysis sometimes seems to be based on personal bias. The intent of the support measure is to describe how directly or easily a pattern can be implemented in a language using built-in constructs. It does generally *not* state whether or not a pattern can be implemented in a language at all. The degree of support states to what extent the user of a language is aided by the constructs that are directly available or built into the language. Its scaling typically is trivalent (or in some cases such as [23, 51] quadrivalent) and distinguishes whether a solution provides *direct* (+), *partial* (+/-) or *no direct support* (-) for a pattern ([43], p. 50), based on the amount of constructs needed in a solution. Constructs are the core building-blocks of a language, such as a decision activity or a fork activity. Adjacent concepts, such as variables or correlation sets generally do not count as constructs. Usually, a candidate solution for a pattern that uses only a single construct provides direct support. A combination of two constructs results in partial support and if more than two constructs are needed no direct support is provided. This trivalent degree can be too coarse. For example, consider the case where a pattern is directly supported in two languages by a single construct. Language A allows to use the single construct in a straight-forward manner and the solution to the pattern is complete. In language B, the single construct needs to be used and a complex configuration of the construct is necessary, consisting of, say, three changes to the default values of its attributes which may be interdependent on each other. Furthermore, the creation of a variable in the process model is also needed. Obviously, the solution in language A is more direct than the solution in language B. Still, they are equal concerning their degree of support.

The aim of this study is to tackle the problems of comparability and selectivity of pattern-based analysis by proposing several improvements and use these improvements to assess the orchestration languages WS-BPEL and WF. The improvements proposed are a unified approach for determining the degree of support a given solution provides for a pattern and a new measure for its calculation. The applicability of the approach is verified by an analysis of the support of the languages for several pattern catalogs. This analysis demonstrates that the proposed approach allows for more well-founded comparison of the two languages than preceding methodologies. The target domain for the analysis is B2Bi. This means that the pattern catalogs used in the analysis are to be selected concerning their relevance for B2Bi. Using the analysis, it is possible to judge the appropriateness of the two languages for the B2Bi domain.

As a foundation for the analysis, in the following section, first the terminology touched here is discussed more extensively and the relevant aspects of the languages in focus are described in detail. A review of relevant pattern catalogs and a methodology for ordering and selecting these catalogs based on their importance for B2Bi succeeds in section 3. Section 4 discusses the approach that aims at improving pattern-based analysis. Parts of this approach have already been published in [24]. In section 5, this approach is used to perform the analysis for the selected catalogs and the implication of the result are discussed. The process models developed during the analysis are available at `http://www.uni-bamberg.de/en/pi/bereich/research/software-projects/pattern-based-analysis-of-orchestration-languages/`. Finally, section 6 ends the study with concluding remarks.

2 Orchestration Languages

Today, orchestration languages are the primary means to describe centrally coordinated, service-based processes [33]. As indicated in section 1, they relate to a special stage in B2Bi. The following section clarifies basic orchestration and B2Bi terminology, as well as fundamental Web Services concepts. Thereafter, the two orchestration languages in the focus of this study, WS-BPEL and WF, and adjacent technologies are examined. As the built-in activities provided with the two languages are of the utmost importance for the following analysis, they will be discussed in detail.

2.1 Fundamentals

All Web Services-based orchestration languages have a common denominator. On the one hand, this is Web Services technology. On the other hand, this is the theoretical background of automated processes in a *Process-Aware Information System* (PAIS), such as typical process elements and structure.

2.1.1 Business-to-Business Integration Schema

The integration of the systems of different enterprises is a complex task. Different groups of people, terminologies and technologies have to work together. A model-driven approach with several stages of development helps to reduce this complexity by introducing several layers of abstraction [41]. In an ideal scenario, service-based B2Bi would follow a top-down development process to ensure consistency throughout several layers of abstraction and different applications. The B2Bi Schema from [41] (illustrated in Figure 1 on the facing page) shows this idealistic relationship of several layers of abstraction for B2Bi. The starting point is the *real world* in which businesses interact. As the name suggests, the real world represents no formalized model, but the everyday interactions among enterprises. To provide automated support for what happens in the real world, a formalization of these interactions is necessary. The first and most abstract model in this approach is the *business model*, which describes the exchange of values among the integration partners [15]. Typically, this model is crafted by domain experts and still rather informal. To characterize the interactions that take place, this model needs to be refined into one or more *business process models*. Each of these models describes the flow of a business process on an abstract level [15]. It might involve multiple autonomous partners and describes their interactions. The next refinement step crosses the border from a mainly domain-oriented view on the interactions to a more technical view. In this step, *choreography* models are developed. Just like business process models, they describe the flow of a business process from a global, partner-independent point of view [33]. However, they add a detailed technical configuration of the flow of control of the process and the publicly visible message exchanges among the different partners, involving message types and Quality of Service (QoS) configurations. A choreography model is supposed to be machine-processable ([40], p. 2). By specifying the publicly visible structure of the interactions, choreography models serve as a means for a detailed agreement among the different partners.

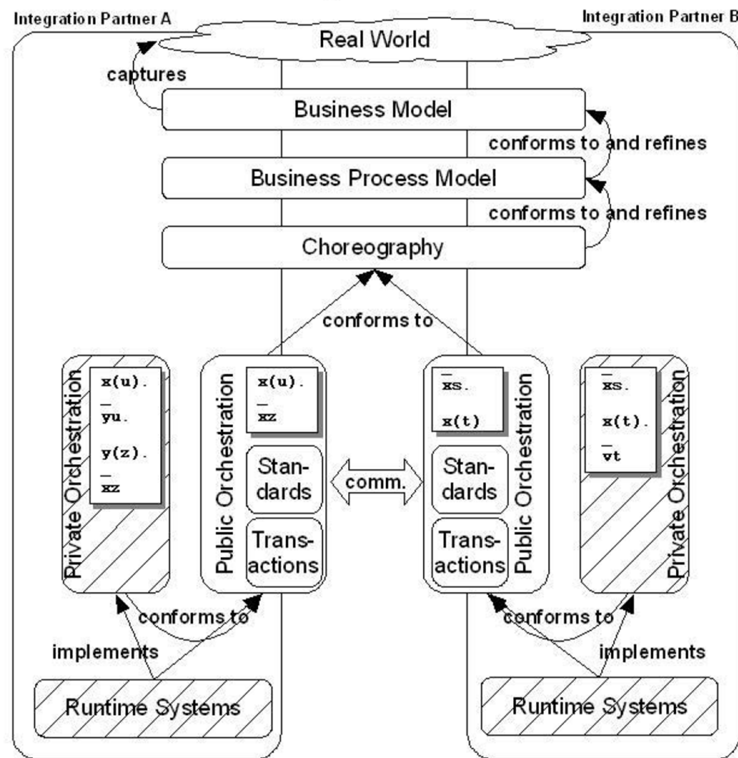


Figure 1: B2Bi Schema as defined in [41]

The next transition is from choreography to *orchestration* models. This is basically a partitioning of the global model into several, local and partner-specific models. *Public orchestrations* can be derived in this way and define the contracts and publicly visible process interfaces of each of the partners ([40], p. 2). At this point, the development process crosses the border from specification artifacts to executable artifacts. All partners implement their public orchestrations in the form of *private orchestrations*, that must conform to the behavior specified in the public orchestrations. Nevertheless, private orchestrations may be structurally different and enrich the process flow with only internally visible details and message exchanges. Private orchestrations are executed by *runtime systems* and integrate preexisting systems and services ([40], p. 2). The last layer, consisting of public and private orchestrations, is built using orchestration languages. These languages are in the focus of this study.

2.1.2 Process Structure and Service-oriented Processes

Orchestrations essentially are centrally-coordinated service-based processes [33]. Consequently, their structure is similar to other centrally-coordinated, but not necessarily service-based processes, called workflows. Due to the importance of such processes, various denominations for the elements of this process structure can be found. Still, all these denominations share a common meaning.

Generally speaking, any information system that provides support for the explicit realization of processes is called a *Process-Aware Information System (PAIS)* ([34], pp. 8/9). These systems

comprise languages for describing processes as well as engines and execution environments. Several elements are essential to the structure of an automated process ([37], pp. 355/356). The basic building blocks of a process are *tasks*, or *activities*. The structure of a process is defined in a process *model* or *schema* which characterizes the control- and data-flow dependencies between several activities. Most process languages today describe the control-flow of process models either as a directed graph or in a block-structured form [21]. A concrete process that is executed according to a process schema is called a *case* or *process instance*. Multiple instances of the same process schema can be executed simultaneously and should run independently of each other. During the execution of a process instance, each of the activities defined in the process schema is also instantiated as an *activity instance* and executed as defined. An activity characterizes a single unit of work and there are three basic types of activities: *atomic*, *block* or *multiple instance* activities. Atomic activities have a simple, self-contained definition. Block activities are more complex. They correspond to the execution of another process, also called *sub-process*. Finally, multiple instance activities mark multiple concurrent executions of an identical activity definition. This structure is outlined in Figure 2.

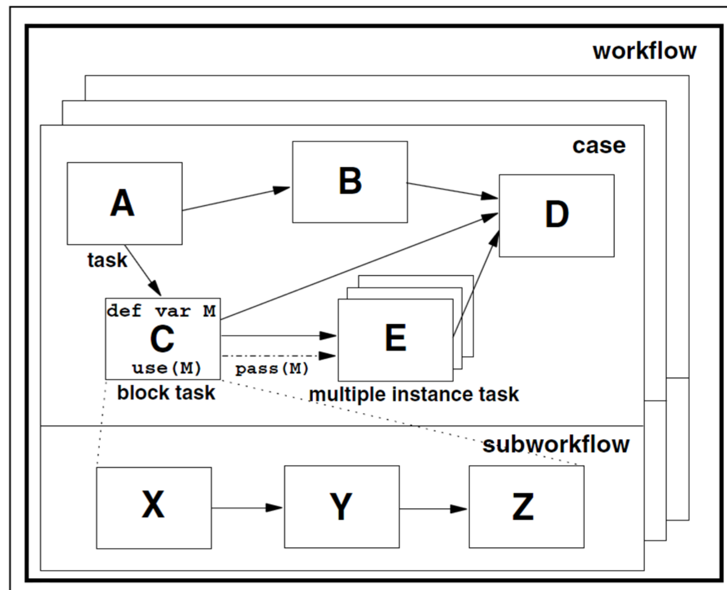


Figure 2: Workflow Structure taken from [37] using workflow terminology

In a service-oriented world, services are ideally suited to implement more complex activities and processes [33]. Here, the execution of a complex activity corresponds to the invocation of a service. In fact, orchestrations frequently are deployed as Web services and run in an engine. For each process schema, a service interface is provided that allows to interact with the process instances. This interaction is accomplished by transmitting messages and the creation of new process instances is triggered through incoming messages. For one service interface, there can be multiple concurrently executing process instances, as described above. Consequently, there has to be a way for an engine to direct incoming messages for a service interface to the concrete process instance for which the message is intended. This is called *message correlation* ([31], pp. 74 ff.). How message correlation can be achieved is specific to a language. The most common way is content-based correlation. Here, an incoming message contains one or more elements

that allow to identify the process instance for which the message is intended. Based on these identifiers the engine is able to direct the message to a corresponding activity of the matching process instance.

2.1.3 Web Services Technology

Today, Web Services are the key technology for implementing SOAs. Web Services and surrounding technologies are promoted by the *World Wide Web Consortium* (W3C). Consequently, the orchestration languages at hand are tightly integrated with this technology. Web Services come in the context of a large array of different standards and technologies the most important of which are the *Web Services Description Language (WSDL)* and *SOAP*.

A Web service is a "software system designed to support interoperable machine-to-machine interaction of a network" ([47], p. 7). Its description language is WSDL and it interacts via SOAP. WSDL [46] is an XML dialect. Its most recent version is 2.0, which was published in 2007. So far, this version is rarely supported by product vendors or integrated in other standards. Also WS-BPEL and WF rely on its preceding version, 1.1. Therefore, this paragraph describes WSDL 1.1. A WSDL `definitions` element characterizes a Web service and can be said to consist of an interface and an implementation part. The interface part first lists `imports` of related files, such as XML Schema Definition (XSD) files or related WSDL files. Then the `types` element describes the XML messages that are used when communicating with the Web service. This can be done by referencing imported XSD files, by defining a schema inside the `types` element or by a combination of both. The next part is the `portType`, which lists the `operations` that are provided by the Web service. These `operations` have a name and may define `input`, `output` and `fault` messages based on the schemas defined in the `types` element. Next follows the `binding` that maps each `operation` in the `portType` to a transport protocol, typically being SOAP via the Hypertext Transfer Protocol (HTTP). Finally, the implementation of the Web service comprises a `service` element which lists one or more `ports`, each of which contains an address under which the Web service can be reached.

SOAP [48], formerly known as Simple Object Access Protocol, is an XML-based messaging framework currently available in version 1.2. A SOAP message consists of an `envelope` that comprises a `header` and a `body`. The `header` is optional and can be used to determine how the recipient of a message should process its `body`. The `body` contains the XML elements that form the payload of the message, which are defined in the WSDL `types` element. As done in the `binding` part of a Web service, when using SOAP, a mapping to a transport protocol has to be defined. Today, SOAP messages are generally transmitted via HTTP or the Simple Mail Transfer Protocol (SMTP), but also other bindings are available.

2.2 Web Services Business Process Execution Language 2.0

The *Web Services Business Process Execution Language*, commonly abbreviated as BPEL, or WS-BPEL is an XML dialect for describing business processes based on Web services in a mainly block-structured manner. The language originated in 2003 as a combination of XLANG by Microsoft and the Web Services Flow Language by IBM. Today, it is promoted by the Organization for the Advancement of Structured Information Standards (OASIS) and since April 2007 it is available in version 2.0 [31]. It builds on WSDL 1.1, SOAP and other Web Services standards. WS-BPEL is the orchestration language that has received the widest interest in business and academia so far and today has become a de-facto standard for implementing orchestrations.

2.2.1 Basic Concepts of WS-BPEL

WS-BPEL comes in two flavors, abstract processes and executable processes ([31], pp. 147–163). As the name suggests, executable processes are fully specified processes that can be deployed in an engine. An abstract process allows for using all the structures that an executable process can use, but permits several of them to be opaque. A process is described in a WS-BPEL process file and is purely Web Services-based. At least one WSDL file has to be imported into the process to serve as an interface for it. Together, these files can be deployed on a WS-BPEL engine. Such an engine provides a Web service reachable at the addresses specified in the `port` elements. The engine allows for the creation and management of process instances and the communication with these instances. WS-BPEL processes may support truly parallel processing of concurrent control-flow branches ([31], pp. 102 ff.).

Apart from the description of the process flow which uses WS-BPEL activities as discussed in the next section, a WS-BPEL process contains several essential elements. The structure of these components is outlined in listing 1. The complete process is contained in a `process` element which forms the global `scope`.

Listing 1: Exemplary Structure of a WS-BPEL Process

```
1 <process name="SampleProcess">
2     <import location="MyRole.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
3     <partnerLinks>
4         <partnerLink name="MyRolePartnerLink" partnerLinkType="MyRolePartnerLinkType"
              myRole="myRole"/>
5     </partnerLinks>
6     <variables>
7         <variable name="InputParameter" messageType="InputMessage"/>
8     </variables>
9     <correlationSets>
10        <correlationSet name="CorrelationSet" properties="PropertyFromWSDL"/>
11    </correlationSets>
12
13    <sequence name="MainProcessFlow">
14        <receive name="StartProcess" createInstance="yes" variable="InputParameter"
              partnerLink="MyRolePartnerLink" operation="OperationFromWSDL" />
15        <!--More basic and structured activities-->
16    </sequence>
17 </process>
```

`partnerLinks`: `partnerLinks` define the relationship between the process and external Web services ([31], pp. 36–39). Each `partnerLink` relates to one partner who parti-cipates in the process. It references a `partnerLinkType` which is to be found in the WSDL file that describes the interface of the partner. At least one WSDL interface needs to be in place and one `partnerLink` needs to be defined for an executable process. This minimum `partnerLink` describes the role of the process itself (has the `myRole` attribute set), so that it can be invoked externally. Other Web services that are invoked by the WS-BPEL process also need to be added as a `partnerLink` and their WSDL description must be imported into the process.

`variables`: A process may contain a set of `variables` ([31], pp. 45–48). `Variables` are always defined within a `scope` (cf. section 2.2.2) which limits their visibility in the process. There can be three types of `variables`. Either their type is a `messageType` read from a WSDL file, or an XML Schema or Schema element type defined by an imported XSD file. During the execution of the process, `variables` can be referenced by several activities which may assign or read data to or from them.

`correlationSets`: WS-BPEL supports message correlation using `correlationSets` ([31], pp. 74–83). Just as `variables` they belong to a `scope`. A `correlationSet` can be used during messaging activities. It references one or more WSDL `properties` that have a `propertyAlias` defined in the imported WSDL files. `Properties` reference `messageTypes` that are XML simple types, defined in the `types` part of the Web service. The values of these types in incoming messages can then be used by the engine to direct the message to the matching process instance.

Finally, the process contains exactly one activity that represents the main process flow.

2.2.2 WS-BPEL Activities

WS-BPEL comprises a variety of activities that can be combined for describing complex process flows. These are basic activities ([31], pp. 84–97) describing single tasks on the one hand and structured activities ([31], pp. 98–112) that encapsulate basic activities and define the control-flow logic on the other hand. Not contained in this set are `scopes` and `handlers` although they also can be treated as activities and have standard elements and attributes. Some activities allow for the definition of `correlations` and exception handling mechanisms.

`scopes`: A `scope` provides the context for a process. It encloses a single activity that captures the flow of control. It is similar to a structured activity, but may contain several additional elements. Due to its importance, it is dealt with separately in the standard ([31], pp. 115–146). A `scope` allows for the definition of `variables`, `partnerLinks`, `correlationSets` and `handlers`. All these structures are only visible inside this scope. Every `process` is also a `scope`. Other `scopes` can be embedded inside the main activity of the `process`.

`handlers`: There are four different types of `handlers` that can be attached to a `scope`. These are `eventHandlers`, `faultHandlers`, `terminationHandlers` and `compensationHandlers` ([31], pp. 118–143). `CompensationHandlers` relate to the concept of transactions. They provide *quasi-atomicity* ([3], p. 313) for a process. In case a process is subject to a transaction and this transaction needs to be aborted, a `compensationHandler` contains

the logic for undoing or compensating the effects of the transaction (as opposed to a roll-back). It is invoked using a `compensateScope` or `compensate` activity which can only be used in fault, termination or compensation handlers. A `faultHandler` is used to react to faults that have been thrown during the execution of an activity. Faults may be thrown in exceptional circumstances or violations of the WS-BPEL standard. A `faultHandler` contains one or more `catch` elements and an optional `catchAll` element which are evaluated in the order of their definition. The activity specified in the first `catch` of which the `faultName` attribute matches the name of the fault that has been thrown, is executed. If it does not match any of them, the activity contained in the `catchAll` is executed. If the fault is not handled in the current `scope`, then it is propagated to the enclosing `scope`. In case a fault is thrown, all activities within the `scope` are terminated and an attached `terminationHandler` is executed. A `terminationHandler` contains the activities that should be performed in case a `scope` needs to be terminated and before the `faultHandlers` are invoked. Finally, an `eventHandler` provides a means to react to messaging or time-related events, such as the reception of an incoming message or expiration of a timer, in parallel to the normal execution flow of a process. Incoming messages are handled by an `onEvent` activity which works similar to a `receive` activity and may trigger the creation of new process instances. In case an incoming message matches the specification of an `onEvent` activity, the message is received and the activities contained in the activity are executed. An `onAlarm` activity can be used to enforce time constraints. The specification of these time constraints works in the same manner as for the `wait` activity. The handler contains an activity that is executed in case the duration or deadline has expired.

Basic Activities ([31], pp. 84–97):

`invoke`: An `invoke` activity calls a Web service. It requires the specification of the `partnerLink` that is to be invoked, as well as the `operation` to be performed and possibly the specification of the `portType`. Depending on the nature of the `operation`, optional input and output variables can be specified for transmitting parameters or storing the return value of the call. As the `invoke` activity is a messaging activity, `correlations` can be defined, as well as exception handling mechanisms using `catch`, `catchAll` or `compensationHandler` activities.

`receive`: The `receive` activity is the counterpart to the `invoke` activity. It allows for the invocation of the process through its Web service interface. For that reason, the `partnerLink` that represents the WS-BPEL process, as well as the `operation` invoked needs to be set. If the operation requires input data, a `variable` needs to be referenced to store this data. A `receive` activity can create a new process instance. If this is necessary, the `createInstance` attribute needs to be set to `yes`. This activity also allows for the specification of `correlations`.

`reply`: The `reply` activity can be used for synchronous interaction with the WS-BPEL process. It sends the reply to a previous inbound messaging activity, such as `receive` or `onMessage`, and thereby answers to a client waiting for this answer. The link to the inbound messaging activity is established via the `partnerLink` and `operation`.

`assign`: The `assign` activity provides a means for copying data elements from and to variables. For this, it contains several `copy` elements which in turn contain `from` and `to` elements

that specify source and target of the copy operation. These elements may contain variable references or expressions. WS-BPEL requires the support for the XML Path Language (XPath) 1.0 [45] as expression language.

`throw`: Internal process faults can be signaled using the `throw` activity. These faults can then be dealt with by a `faultHandler`

`wait`: The `wait` activity can be used to delay the execution of the process. This can either be done by using a specific amount of time in the `for` element or a date that serves as deadline in the `until` element.

`empty`: The `empty` activity can be used for doing nothing. This can be necessary as a placeholder or for satisfying semantic constraints.

`extensionActivity`: This activity can be used to integrate non-standard activities into a WS-BPEL process. It may contain custom constructs that can be interpreted by a specific engine.

`exit`: The `exit` activity terminates a process instantly without permitting any termination, fault or compensation handling.

`rethrow`: The `rethrow` activity can be used inside a `faultHandler`. It rethrows the specified fault, ignoring changes made to the original fault data by the `faultHandler` in the interim.

Structured Activities ([31], pp. 98–114):

`sequence`: The `sequence` activity can comprise a number of other activities which are executed sequentially in the order in which they appear in the `sequence`.

`if`: This activity allows for the selection of one among a set of several control-flow branches to be executed. It contains a `condition` which must be a boolean expression and an activity that is executed in case the `condition` evaluates to true. Optionally, a number of `elseIf` elements can be defined. Their structure resembles that of the `if` and they are executed in case the condition of their predecessors evaluates to false. Finally, an `else` element may be contained at most once and comprise an activity that is executed in case no `condition` in the entire `if` activity evaluates to true.

`while`: One way of defining a loop is the `while` activity. Like the `if` activity, it contains a `condition` and a body activity. This activity is executed as long as the `condition` evaluates to true. The `condition` is evaluated each time before executing the activity.

`repeatUntil`: Another way of defining a loop is the `repeatUntil` activity. It also contains a `condition` and an activity. Here however, the activity is executed as long as the `condition` evaluates to false. This is checked each time after executing the activity.

`forEach`: A third way of implementing a loop is the `forEach` activity. It contains a `scope` which is executed a given number of times. This number is determined by the `finalCounterValue` and `startCounterValue` of the `forEach` activity and cannot be changed once the `forEach` activity has been initialized. The execution of the different instances of the `scope` can take place in sequential order or in parallel. An optional `completionCondition` can be used to end the loop prematurely. This condition is evaluated each time one of the instances of the `scope` completes and in case it evaluates to true, all `scopes` that are still executing are canceled and the `forEach` activity completes.

`pick`: The `pick` activity is used to react to events. It must contain at least one `onMessage` activity. The body of this activity comprises an activity which is executed in case the specified message is received. The `pick` then executes the activity in the `onMessage`

where the specified message is received first. The `pick` may also contain one or more timers represented by `onAlarm` activities.

`flow`: The `flow` activity can be used for defining concurrent execution of multiple branches. It contains several child activities all of which can be executed in parallel. The `flow` completes when all its children have completed. It also allows for the specification of `links` between its children. `Links` allow to define precedence relationships between the activities in the `flow`. Activities may act as `sources` or `targets` of `links`. Whether or not `links` are activated and the target activities are executed may depend on conditions. Although `links` break the block-structured modeling style of WS-BPEL, they do not allow for arbitrary unstructured process models. Link semantics are quite strict and as an example, `links` may not create cycles ([31], pp. 105–112).

2.3 Windows Workflow 4

Unlike WS-BPEL, WF is not an open standard but is developed by Microsoft as a part of the .NET framework [26], currently available in version 4. All discussion and analysis in this study builds on the language version available in October 2010. This is basically version 4.0 (for all further discussion simply version 4), as relleased with .NET 4.0. The .NET framework is a software framework for developing and executing applications for the Windows platform. Core to it are two components, the *Common Language Runtime (CLR)* and the .NET class library. The CLR is a virtual machine that allows to execute *Common Intermediate Language (CIL)* code ([26], section "Common Language Runtime (CLR)"). In .NET, applications can be written in different languages, such as C# or Visual Basic. This code is then compiled to CIL code. During the runtime of an application, the CLR compiles the CIL code to the native language of the system using a just-in-time compiler. Based on CIL, it is possible for applications in different languages to interoperate directly. One application might be written in C# and another one in Visual Basic. As both of them are compiled to CIL, the C# application might make use of the Visual Basic assembly and vice versa. The .NET class library comprises a set of base classes available to all languages that can be used in .NET.

Apart from these two core components, .NET comprises various other frameworks and Application Programming Interfaces (APIs) that target specific functionality and development tasks. Examples are the *Windows Communication Foundation (WCF)* for developing service-oriented applications and the *Windows Workflow Foundation*[1] for developing process-based applications. The latter one is in primary focus here. A comprehensive description of WF can be found in [5, 6] or its online documentation[2] [27]. The Windows Workflow Foundation was first introduced in 2006 with .NET 3.0 and a revised version was shipped with .NET 3.5. .NET 4 contains a completely reworked version of the underlying language of WF with major changes to its structure. The base activities shipped with it have been completely rewritten. In April 2011, Microsoft shipped its first platform update of .NET 4[3]. This platform update also introduces changes to WF, most notably the re-introduction of an additional modeling style. The following discussion and analysis, does not consider this platform update, but is based on the initial version of WF contained in .NET 4.

2.3.1 Basic Concepts of WF

The basic structure of workflow-based applications can be seen in Figure 3 on the following page[4]. Any .NET application can serve as host process for a workflow. Such an application needs to contain the WF runtime engine which provides support for executing workflows. Runtime

[1] The term Windows Workflow Foundation is used for the overall technology and part of .NET. Windows Workflow on the other hand describes the language which is part of this foundation and is used to develop workflows.

[2] The .NET Framework Developer Center also provides boards, references, and tutorials which are maintained by Microsoft officials and can serve as a helpful reference. Its WF section is available at `http://msdn.microsoft.com/en-us/netframework/aa663328.aspx`.

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

[4] The Figure depicts the architecture of WF 3.5. This architecture has not changed for WF 4 and is still valid here.

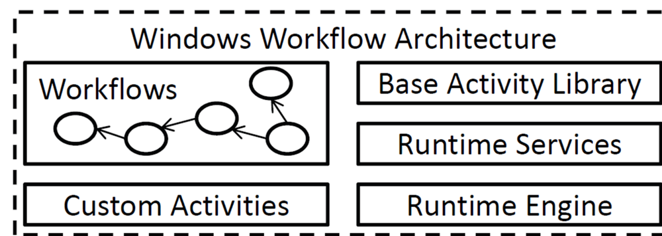


Figure 3: Overview of the Windows Workflow Foundation Architecture taken from [55]

Services add additional maintenance functionality such as tracking or persistence for workflows. The *Base Activity Library (BAL)* comprises the built-in activities that come with WF and that way constitutes the underlying language. Based on these activities, other custom activities can be developed to implement specific functionality. Custom activities can be created in different ways. They can be written in code using one of the traditional languages provided by .NET, such as Visual Basic or C#, or they can be written in eXtensible Application Markup Language (XAML) ([6], p. 18). XAML is an XML dialect used in several components of .NET. WF leverages it as an alternative way to develop workflows. Just as other .NET languages, XAML is compiled into CIL code and thereby interoperable with the other .NET languages.

A workflow essentially is an activity ([5], p. 45 ff.), a unit of work. It is assembled out of other activities and can in turn be used like any other activity in other workflows. This provides a natural mechanism for decomposition and abstraction. In contrast to most workflow languages, WF comes with two modeling styles, the *procedural* or *sequential* modeling style ([5], p. 163 ff.) and the *flowchart* modeling style ([5], p. 229 ff.). These modeling styles determine how the flow of control between activities can be expressed. The procedural modeling style is block-structured. The flowchart modeling style is graph-oriented and uses direct links, called `FlowSteps`, between activities to direct the flow of control. As activities are self-contained, the two styles are interoperable and a workflow may mix them freely. For example a workflow implemented in the procedural style may arrange several activities which are implemented in the flowchart style in sequential order. Figure 4 depicts a graphical representation of a flowchart workflow. Several structured activities allow for the definition of variables and arguments which are also scoped by the activity for which they are defined. Expressions can be used at several points and activities. Up to WF, expressions can only be defined as Visual Basic expressions or in code as expression activities by extending certain classes of the WF API ([5], p. 58 ff.). One specialty of WF is its threading model. As opposed to WS-BPEL, there no truly parallel, but pseudo-parallel processing of concurrent control-flow branches with an activity-level granularity. By default, a workflow is executed on a single thread ([5], p. 73 ff.). Activities can be scheduled to run in parallel which results in a pseudo-parallel interleaving of these activities on the single thread. The structure of this interleaving depends on the nature of the activities to be interleaved. Normally, the activities simply execute on the single thread in the order of their definition. Each time one of the activities completes, the scheduler hands over the control to the next one. If a currently executing activity becomes idle, for example because it waits for an incoming message, the scheduler switches control to the next activity. Ultimately, the scheduler will return the control to the previous activity and if there is not more reason for it to be idle, processing can continue. Multi-threaded execution is still possible by using custom activities that explicitly require to be executed on a separate thread. These

activities can only be declared in code by inheriting from the class `AsyncCodeActivity`. This also means that multi-threaded execution cannot be achieved by only declaring activities in XAML and relying on the base activity library[5].

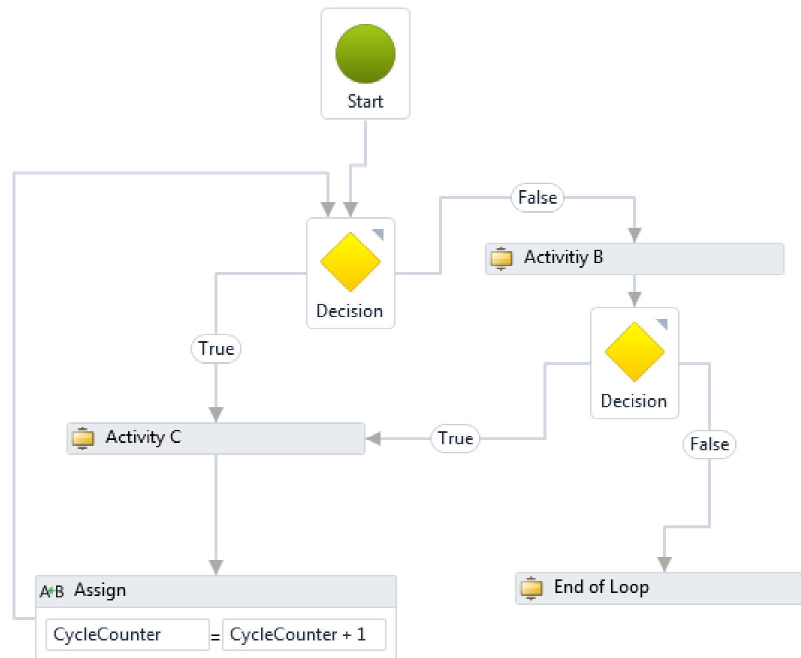


Figure 4: Graphical example of a workflow in the flowchart modeling style

To be applicable as an orchestration language in a Web Services-based SOA, communication with Web Services must be supported. In WF, this is achieved with the help of WCF ([5], p. 313 ff.). Using WCF, a workflow can be promoted as a Web service[6], in the terminology of WF called *workflow service*. A workflow service essentially forms an orchestration. The main differences between ordinary workflows and workflow services is that workflow services are directly deployable as Web services, but can no longer be used as an activity in another workflow. Apart from this, they differ very little in their structure. Workflow services must contain at least one inbound messaging activity that creates an instance of the workflow, for example a `Receive` activity with its `CanCreateInstance` attribute set to true. When implemented in XAML, the top level element of the workflow file changes to `WorkflowService` instead of `Activity` and the workflow file must be a `.xamlx` file instead of a `.xaml` file. Just like WS-BPEL, WCF relies on WSDL 1.1. The structure of the WSDL `definition` that forms the interface for a workflow service can be inferred from the messaging activities of the workflow service and the data types used in these activities. For example, the `OperationName` of a `Receive` activity is mapped to an `operation` in the WSDL `definition`. A workflow or workflow service can also act as a Web service client using `Send` activities. The use of WCF is transparent when developing workflows.

[5]There is one exception to this. The base activity `InvokeMethod` can be executed on a seperate thread.

[6]In WF 4, as opposed to WF 3.5, the reverse is not directly supported. There is no built-in feature that allows a workflow service to implement a given WSDL definition.

All necessary configurations can be made in XAML.

Finally, several classes of the .NET class library are also used in BAL activities and therefore essential for WF. First, these are time-related classes, such as `TimeSpan` or `DateTime` which are for example used in `Delay` or `TransactionScope` activities. Second, these are data type related classes, such as `Collection`. Third, message correlation in WF is achieved using correlation-related classes, such as `CorrelationHandle` for storing correlation identifiers at run-time.

2.3.2 Base Activity Library

The Base Activity Library of WF contains a large set of activities which is considerably bigger than the set of activities provided by WS-BPEL. These activities are grouped in nine categories: Primitives, control-flow, flowchart, messaging, runtime, transactions and compensation, collection management, error handling and migration ([5], p. 103 ff.). The primitives category groups several activities that perform basic operations. Similarly, the collection management category bundles basic operations for using collections. The control-flow category comprises activities needed to direct the flow of control in the procedural modeling style and the flowchart category contains the activities needed for the flowchart modeling style. Activities for external service-based communication can be found in the messaging category. Runtime activities concern the persistence and termination of process instances. Support for transactions in workflows is accomplished by activities from the transactions and compensation category. Activities for exception handling can be found in the error handling category and finally the migration category contains an activity for executing WF activities from older versions in a WF 4 workflow. In the following, each of the activities is explained.

Primitives ([5], pp. 107/108):

`Assign`: This activity sets the value of a single variable. The value may be a literal or an expression and it must conform to the type of the variable.

`Delay`: The `Delay` activity can be used to stall a workflow for a given amount of time. This amount can be defined as a fixed literal or as an expression of type `TimeSpan`. When the timer expires, the execution of the activity completes and succeeding activities may be executed. During the execution of a `Delay` activity, a workflow is idle and may be unloaded from memory.

`InvokeMethod`: This activity can be used to invoke a method on an object or class using input parameters if necessary and obtain a result. The target object or class can be determined using an expression or can be contained in a `Variable` available in the workflow.

`WriteLine`: `WriteLine` writes a String which can be a literal or the result of an expression to a `TextWriter`. By default, the output is directed to the console.

Control-flow ([5], pp. 104/105):

`Sequence`: This structured activity can have an arbitrary number of child activities and schedules them for execution in the order of their definition.

`If`: The `If` activity requires an expression which returns a boolean value as condition and may have two child activities. One for its `If.Then` block and one for its `If.Else` block.

If the condition expression evaluates to true, the activity contained in the `If.Then` block is executed. Otherwise, the activity contained in the `If.Else` block is executed.

`Switch<T>` ([5], pp. 164/165): Like the `If` activity, the `Switch<T>` activity takes an expression as condition. However, it allows to define multiple child activities, each one as a single case. This activity is generic, so the type of the expression value needs to be fixed when defining it. Each case is then combined with one possible expression value as guard and on execution of the `Switch<T>` the child activity where the guard matches the result of the expression is executed. The cases are evaluated in the order in which they are defined. It is also possible to define a default case that is executed when none of the guards of the other cases matches. At all events, only one of the cases is executed. The `Switch<T>` cannot trigger multiple branches of execution.

`While`: The `While` activity implements a loop. It takes a boolean expression as condition and an activity as body. The body activity is then executed as long as the expression evaluates to true. The value of the condition is checked each time before executing the activity.

`DoWhile`: The `DoWhile` activity is very similar to the `While` activity. The only difference is that the expression is evaluated each time after executing the body activity. This way, the body activity is executed at least once.

`Parallel` ([5], pp. 176–178): This activity takes multiple child activities and schedules all of them for immediate execution. As the WF execution model is single-threaded, this does not mean that the activities are executed at the same time. Instead, the execution of the `Parallel` activity results in an interleaving of its child activities. In case the currently executing activity gets idle, for example because it contains a `Delay` activity or waits for an incoming message, the control switches to another activity. The `Parallel` activity also allows for the specification of a `CompletionCondition` which is a boolean expression. Normally, the `Parallel` activity would complete when all its children have completed. In case a `CompletionCondition` is specified, this condition is evaluated each time after one of the child activities completes. If it evaluates to true, all child activities that are still executing are canceled and the `Parallel` activity completes.

`ForEach<T>` ([5], pp. 195/196): Another way of implementing a loop is the `ForEach<T>` activity. It takes a collection of values and another activity as body. The body activity is then executed for each of the values in the collection. Again, this activity is generic, so the type of the values in the collection needs to be fixed upon definition.

`ParallelForEach<T>` ([5], pp. 195/196): This activity combines the `Parallel` activity and the `ForEach<T>` activity. Just like the `ForEach<T>` activity, it takes a collection of values and an activity as body and executes this body activity for each of the values. The difference lies in the scheduling of the execution of the body activity. While the `ForEach<T>` schedules the execution of the body activity sequentially for each of the values, the `ParallelForEach<T>` immediately schedules the body activity for execution for each of the values and this execution may be parallel. Just as with the `Parallel` activity, this does normally not result in a true concurrent execution, but in an interleaving. Also, the `ParallelForEach<T>` allows for the specification of `CompletionCondition` which behaves in the same manner as for the `Parallel` activity.

`Pick` and `PickBranch` ([5], pp. 302–304): These activities in combination allow for an event-based direction of the control-flow. The `Pick` activity contains several `PickBranch` activities, each of which encapsulate an event. A `PickBranch` again contains two child activities, a trigger and an action. All trigger activities start executing in parallel on

the initialization of the Pick activity. On completion of the first trigger activity, all PickBranch activities are canceled except for the one of which the trigger activity has completed. Then, the action activity of this PickBranch is executed. When this activity completes, also the Pick activity completes.

Flowchart ([5], pp. 105/106):

Flowchart: As its name suggests, this activity is the basic building block for the flowchart modeling style and contains a start node and an arbitrarily long nesting of FlowSteps. Each FlowStep contains an activity and a reference to or definition of the element (activity or FlowStep) to which the activity is pointing.

FlowDecision: This activity implements a simple branch in a flowchart activity. It contains an expression that returns a boolean value. Depending on the outcome of the expression, the FlowDecision routes the flow of control into one of the alternative directions, similar to the If activity.

FlowSwitch and FlowSwitch<T>: If there are more than two directions the flow of control can take in a flowchart workflow, the FlowSwitch activity needs to be used. This activity contains an expression and allows to define multiple links based on certain values of the expression. The activity comes in a generic and non-generic version. While in the non-generic version the value of the expression may only be a String, the generic version allows for any other types.

Messaging ([5], pp. 317–326):

Send: The Send activity is used to invoke an external service via WCF from within the workflow. This requires the setting of several parameters which is largely dependent on the service invoked. As a minimum, the ServiceContractName and OperationName need to be set. Generally, also the Endpoint to which the message should be transmitted needs to be identified by an AddressUri and a Binding. The activity also allows for passing parameters and initializing CorrelationHandles.

ReceiveReply: This activity can only be used in combination with the Send activity and is used to implement a synchronous Web service operation. Therefore, it needs to reference the Send activity it belongs to. It blocks until it receives a message that is sent as a reply to the invocation of a synchronous operation with the Send activity.

Receive: The Receive activity is the counterpart to the Send activity. It can be promoted as a Web service operation via WCF. This way, clients can send SOAP messages to the workflow. It requires the definition of the OperationName to be promoted by the service and also the ServiceContractName needs to be fixed. The reception of a message through a Receive activity may create a new workflow instance. The Receive activity may also be used to initialize a CorrelationHandle.

SendReply: Just like the ReceiveReply activity accomplishes synchronous communication for the Send activity, the SendReply does the same for the Receive activity. It needs to be linked to a previous Receive activity and cannot exist in isolation. It allows for the definition of the message content to be sent, as well as for the specification of correlation information. This way, it transmits the response to the invocation of a synchronous operation provided by the workflow.

`SendAndReceiveReply`/`ReceiveAndSendReply`: These two activities are templates for facilitating development. The templates consist of the activities described in their name and a `Sequence` activity which encloses the two. For example, the `SendAndReceiveReply` activity consists of a `Sequence` that comprises a `Send` activity followed by a corresponding `ReceiveReply` activity. The templates also provide the definition and configuration of a `CorrelationHandle`. Thereby, the `ReceiveAndSendReply` activity maps to a synchronous Web service operation.

`CorrelationScope`: The `CorrelationScope` activity serves as a container for messaging activities. It contains an expression that identifies the `CorrelationHandle` object that provides the correlation information for the scope. All messaging activities that are children of the scope use this `CorrelationHandle`.

`InitializeCorrelation`: `CorrelationHandles` can not only be initialized in the context of messaging activities, but also at any point in the workflow using this activity.

`TransactedReceiveScope` ([5], p. 412): This activity is related to the the `TransactionScope` activity described below. It allows to trigger a transaction in the workflow. The trigger is an incoming message consumed by a `Receive` activity. The body of the `TransactedReveiceScope` activity contains an activity that is to be executed with a transaction. This transaction is triggered by a `Receive` activity that is placed in the `Request` part of the `TransactedReceiveScope`. This transaction is not visible from outside, i.e. the `TransactedReceiveScope` is not sufficient to mark a workflow as a resource in the sense of a distributed transaction protocol, such as the *two-phase commit protocol* [18].

Transactions and Compensation:

`TransactionScope` ([5], pp. 540/541): The `TransactionScope` activity groups activities that should be surrounded by a transactional context. These activities are declared as the body of the scope. If all activities inside the body complete successfully, the transaction is committed. If one of the activities throws an exception that is not handled inside the scope, it is disposed and all activities inside the scope are rolled back. The `TransactionScope` can also be used to grant atomic read and write access to shared variables in the workflow. Altogether, this activity allows to implement atomic and isolated transactions that can be rolled back in the context of a single workflow instance. It also allows for the specification of a timeout. If the timeout is exceeded by the body activity, the transaction should be rolled back ([5], p. 540). This does not mean that the transaction is immediatly stopped when the timeout is exceeded. When the `TransactionScope` eventually completes, the duration of the execution is compared to the timeout. If it is bigger, the transaction is considered to have failed and is rolled back. This behavior implies that a timeout is not sufficient to stop a deadlocking transaction.

`CompensableActivity` ([5], pp. 549/550): This structured activity encloses an activity that can be compensated. It allows for the definition of compensation, cancellation or confirmation handlers. These handlers are normal activities that are executed if they are triggered through the body activity.

`Compensate` ([5], p. 108): This activity triggers the execution of the compensation handler of the `CompensableActivity` it is contained in. Confirming the activity is then no longer possible.

`Confirm` ([5], p. 108): Similar to the `Compensate` activity, this activity activates the confirmation handler of a `CompensableActivity`. Cancellation is no longer possible.

`CancellationScope` ([5], p. 564): This activity works similar to the `TransactionScope` activity. It contains an activity as body and allows for the specification of a handler activity that is executed in case the body activity is canceled.

Runtime:

`Persist` ([5], p. 417): A workflow is persisted (and unloaded from memory) when it becomes idle, for example through the execution of a `Delay` activity or while waiting for an incoming message. This can also be triggered explicitly using the `Persist` activity.

`TerminateWorkflow` ([5], p. 107): As the name suggests, this activity terminates a workflow and optionally allows to specify a reason for the termination and an exception to be thrown. All activities executing at the time of termination are immediately canceled.

Collection Management ([5], pp. 198–199):

`AddToCollection`: This activity requires the specification of an item and a collection whose contents are of the same type as the item. It then adds the item to the collection.

`RemoveFromCollection`: The counterpart to the `AddToCollection` removes an item from a collection.

`ExistsInCollection`: This activity tests whether an item is contained in a collection.

`ClearCollection`: This activity removes all elements from a collection.

Error Handling:

`TryCatch` ([5], pp. 530/531): Activities that might throw exceptions can be included in a `TryCatch` activity. Apart form its try block, this activity allows for the specification of a collection of `Catches` as well as a `Finally` element. A `Catch` is a mapping of an exception type to an activity. In case the try block of the `TryCatch` activity throws an exception, the `Catches` are checked in the order of their declaration. The activity contained in the `Catch` whose exception type first matches the type of the exception thrown is executed. Ultimately, the activity in the `Finally` element is executed regardless of the outcome of the `Try` or `Catch` elements.

`Throw` ([5], p. 109): The `Throw` activity allows to raise an exception and propagate it to the activity enclosing the `Throw`.

`Rethrow` ([5], p. 109): This activity can only be used within the `Catch` element of a `TryCatch` activity. It re-raises the exception handled by this `Catch` and propagates it to the enclosing activity.

Migration:

`Interop` ([5], p. 110): WF is not backwards compatible with older versions of the language. Therefore, the `Interop` activity may serve as container for activities that were created with an older version of the language. These are executable in a WF 4 workflow with certain restrictions.

3 Patterns for Orchestration Languages

The notion of patterns was first introduced for the architecture of physical structures by Alexander [1]. Years later, the *gang of four*[7] introduced this concept to logical architectures, namely software designs [17]. Since then, patterns have gained noticeably interest in computer science and have become ubiquitous in any area that concerns the construction or design of systems, including the construction of processes and services. Patterns describe an abstract and elegant solution to commonly reoccurring problems. In other words, patterns are heuristically derived best practices. By using such heuristics, it is possible to speed up the development of systems and enhance the quality and robustness of the outcome ([17], p. 1/2). The method of pattern-based analysis uses patterns for a slightly different purpose. Instead of using them as exemplary solutions to common problems, they are used as a means for comparing the expressiveness of languages [43]. Here, patterns capture features and constructs that should be present in a process language. A language can then be assessed whether or not it does support these features and constructs.

There are numerous patterns relevant in the context of SOAs. In general, these patterns are grouped into pattern catalogs based on their nature. Many of them can also be considered to be relevant for analyzing orchestration languages. The following section identifies pattern catalogs relevant to the area of processes in SOAs with respect to B2Bi. Not all of these catalogs are equally appropriate for pattern-based analysis. Consequently, a selection of catalogs has to be made and the most important catalogs are to be used in the analysis of this study. Therefore, the following sections present an approach for selecting catalogs and a ranking of catalogs resulting from this approach. Finally, an overview of the current state of pattern-based analysis of WS-BPEL and WF is provided.

3.1 Survey of Pattern Catalogs

A total of fourteen pattern catalogs has been identified by searching relevant scientific literature, conference proceedings, and journal volumes of recent years. As SOAs, orchestration languages, and B2Bi are current topics in academic research, it is likely that new pattern catalogs are developed in the future and also this list is to be extended soon. Three categories of catalogs were identified based on the contents of the catalogs. These categories are *workflow patterns*, *service interaction patterns* and *integration patterns*. The biggest category, the workflow patterns, originated from the area of workflow systems and the seminal publication [43] also introduces the method of pattern-based analysis. The workflow patterns address aspects shared by automated processes or more general by PAIS, no matter whether they are service-oriented or not. The focus on service-orientation is established by the service interaction patterns. Finally, integration patterns focus on the integration of applications in a single enterprise or multiple autonomous enterprises. Obviously, they are highly relevant to B2Bi. They have in common with SOAs, the paradigm of loosely-coupled interaction of autonomous components. Table 1 shows the classification of the catalogs identified.

[7]The authors of [17] are generally referred to as “the gang of four”. Due to the importance of this book for software engineering, this denomination is quite well-known among software architects.

Table 1: Classification of pattern catalogs according to three categories

Workflow Patterns	Service Interaction Pattern	Integration Patterns
Control-flow [38, 43]	Service Interaction [3]	Enterprise Integration [20]
Data [37]	Multi-Party Multi-Message	Process-Oriented
Resource [35]	Request-Reply	Integration [19]
Exception Handling [39]	Conversation [28]	
Change [51]	Correlation [2]	
Time [23]		
Protocol [44]		
Activity [42]		
Process Instantiation [10]		

3.1.1 Workflow Patterns

The seminal publication [43] in the context of the workflow patterns is about control-flow structures in automated processes. The authors identified twenty structures they extracted through the evaluation of case studies, workflow systems, scientific literature, and comparable data sources. In 2006, the original catalog of control-flow patterns was even revised and extended to capture forty-three patterns [38]. The initial control-flow patterns catalog consists of six categories: *Basic*, *advanced branching and synchronization*, *structural*, *multiple instances*, *state-based*, and *cancellation* patterns [43]. Apart from extending this list, the follow-up [38] revises the original patterns with clearer definitions and formal representations for each of the patterns. Several new categories are introduced and the former categorization of patterns is slightly changed. In some cases, the original patterns have been split up into several new ones due to a degree of ambiguity in the initial pattern definition. The new categories introduced by the revised catalog are *iteration*, *termination* and *completion* patterns.

Aside from control-flow, data is considered one of the most important perspectives in workflows ([43], p. 7). In [36, 37], Russell et al. identify forty patterns relevant to the data perspective in workflows. In general, the notion of data is connected to a number of data structures. These structures are not as relevant when looking at abstract problems and solutions concerning the use of data in processes. Here, characteristics like the *visibility* of data elements to activities in process instances, *interactions* based on data, describing how data is communicated between activities, the mechanisms for the actual *transfer* of data between activities and the way in which data directs the *routing* of the control-flow of a process ([36], p. 3) are of primary relevance.

Additionally to control-flow and data, resources are an important perspective of workflows ([43], p. 2). The workflow patterns initiative investigated resource patterns in [35] with special focus on human resources. This catalog comprises more than forty patterns, grouped by seven categories. The *creation* patterns describe restrictions of activities concerning the resources which may execute them and the time and place at which they may be executed ([35], pp. 222/223). *Push* patterns concern the way in which resources are made familiar with the activities they

should execute from the viewpoint of the workflow engine ([35], pp. 223/224). *Pull* patterns consider this aspect from the resources point of view ([35], pp. 224/225). *Detour* patterns discuss aspects that are necessary if normal activity execution is interrupted and for example exceptions need to be handled ([35], pp. 225/226). Event-based creation of activities and their allocation to resources is dealt with by *auto-start* patterns ([35], pp. 226–228). Finally, the *visibility of activities* for resources and the possibility of *multiple resources* working on the same activity is considered ([35], p. 228).

Although process models often focus on best case behavior, the handling of failures is inevitable. Possible failures and errors can be grouped in the form of exceptions for which handling and compensation mechanisms may be specified. Such exception handling mechanisms for workflows are described in [39]. To categorize these patterns, five possible types of exceptions that can occur in a process ([39], p. 291–293) are defined. The authors then characterize the possible states of an activity and the transitions that can be taken, given an exception occurred, in the form of a *life cycle* ([39], p. 293/294). They further describe the exception handling that can be performed at *process instance level* and the *recovery action* that can be undertaken ([39], p. 295). An exception handling pattern is now characterized as a triplet of the transition that is being taken in the life cycle of the activity, the exception handling done at process level and the recovery action that is taken. For each of the exception types, certain patterns are applicable. While there are 135 patterns possible, not all of them are applicable for all exception types and the authors limit the total amount to 108 combinations of pattern and exception type ([39], p. 296).

Frequently changing business partners and the need to optimize processes are a major factor for the relevance of SOAs in B2Bi. Change patterns [51] describe features that should be supported by PAIS to be able to cope with such changes. The authors consider changes of the control-flow of a process at schema and instance level. They distinguish two sets of patterns, *adaptation* patterns and patterns for *predefined changes*. Adaptation patterns characterize features provided by an editor for developing process models and are applied for unexpected needs of change. Patterns for predefined changes describe structures that serve as hooks or placeholders for changes and flexibility during the run-time of a process ([51], p. 448 ff.).

Another important topic in business processes is the management and surveillance of time constraints ([9], p. 950). Time constraints are the prerequisite for the specification of a number of quality of service attributes that are often indispensable when doing electronic business. Also, the control-flow of a process may depend on time constraints. To allow for a systematic comparison of time aspects in PAIS, a set of *time patterns* is proposed in [22,23]. These patterns were discovered by the analysis of a large set of processes from the medical, automotive and aviation industry. The authors identify ten patterns grouped by four categories. These four categories are *durations and time lags*, *restrictions of process execution points*, *variability* and *recurrent process elements* ([23], p. 97).

The aspect of when and how process instances are created are covered by *process instantiation patterns* [10]. Events that determine when new process instances are to be created is covered by *creation patterns*. There may be multiple entry points to a process model. Which of these points may be activated on instance creation is characterized by *activation patterns*. *Subscription patterns* discuss for which start events of a process instance event subscriptions

are created after its instantiation. Finally, *unsubscription patterns* cover the aspect of how long these subscriptions are kept ([10], p. 784 ff.).

Derived from a large set of real-world process models, activity patterns [42] capture reasonable blocks of business functionality that are present in many business processes. Such functionality is for example the *approval* ([42], pp. 97/98) of an object or a *notification* ([42], p. 101). All in all, there are seven such patterns each describing a distinct block of functionality.

The last pattern catalog in the workflow patterns category [44] was published one month before the initial workflow control-flow patterns catalog in 2003. Nevertheless, its topic, contracting workflows and protocol patterns, fits nicely into this category. It concerns the contracting of legal agreements by automated processes. In B2Bi, contracting, like ordering products, is a frequent part of a business process. The author of the pattern catalog divides electronic contracting into four phases, *specification*, *negotiation*, *execution* and *acceptance* ([44], pp. 155/156). The last three of these phases can gain advantages from a pattern language and [44] makes a start by describing eight patterns for the negotiation phase.

3.1.2 Service Interaction Patterns

The first pattern catalog concerning services was published in 2005 [3,4]. The thirteen patterns contained in this catalog were derived from real-world Business-to-Business (B2B) processes. The authors characterize the patterns according to three dimensions ([3], pp. 303/304).

1. The number of parties involved in a service interaction which can be two or unbounded, leading to bilateral or multilateral interactions.
2. The number of message exchanges in a bilateral interaction which can also be two or unbounded, leading to single-transmission or multi-transmission interactions.
3. Whether the recipient of a response in a two way interaction necessarily is the same as the sender of the request, being a round-trip or a routed interaction.

With the help of these dimensions, the authors identify four categories of service interaction patterns, namely *single-transmission bilateral interaction patterns*, *single-transmission multilateral interaction patterns*, *multi-transmission interaction patterns* and *routing patterns* ([3], p. 304).

Just as with the workflow patterns, the initial service interaction pattern catalog has been extended by various other contributions, although there are not yet as many new contributions as for the workflow patterns. One extension is proposed in [28, 29]. The authors describe five distinct pattern families, but without deriving all the concrete patterns which comprise a family. The families they describe involve *multi-party multi-message request-reply conversation* for scenarios where a party interacts with multiple others using multiple messages, *renewable subscription* for long-running interactions and the families of *message correlation*, *message mediation* and *bipartite conversation* which all address aspects of message correlation on different levels of abstraction ([29], pp. 3/4). The family of multi-party multi-message request-reply conversation patterns involves one requester party and a number of responding parties. The requester builds a compound request that consists of a number of request messages to each of

the responders which are sent simultaneously. Although the requester expects a response for each of the request messages, some of the responders may not reply. All responses arriving at the requester are then queued and the requester consumes a subset of the messages arrived and processes a subset of the message consumed ([28], p. 739). There exist a number of configuration options that, according to their setting, produce different patterns. Roughly 3000 distinct patterns can be produced this way ([29], p. 3).

The correlation patterns [2] characterize mechanisms for achieving message correlation. Instead of proposing a conceptual framework for generating patterns as in [28], they describe a concrete set of eighteen patterns, grouped in three categories. The first of these categories regards primitive mechanisms for achieving message correlation, such as *key-based* or *reference-based* correlation. The second category focuses on the behavior that can occur between several processes, such as *conversation overlap*. Finally, the third category comprises patterns that characterize the relationship between process instances and conversations which could for example be *one process instance - many conversations*.

3.1.3 Integration Patterns

In contrast to the workflow or service interaction patterns, the enterprise integration patterns by Hohpe and Woolf [20] do not primarily aim at benchmarking workflow systems or process modeling languages, but are more like the original design patterns from [17]. As the name suggests, these patterns primarily address the problem of integrating and connecting different applications. Furthermore, the authors focus on a message-based integration style, as opposed to a file transfer, shared database or remote procedure invocation integration style ([20], pp. 43–53). From a more abstract point of view, applications can be seen as services integrated through messages and so the relevance of this pattern catalog to service compositions becomes obvious. All in all, the authors present 66 patterns that revolve around all aspects of messaging:

- The design of messaging systems.
- Message channels and their management.
- The construction, routing and transformation of messages.
- The design of the endpoints that send and receive messages.

Many of the patterns presented describe the basic infrastructure needed to communicate in a SOA.

Analogously, the patterns for process-oriented integration [19] describe generic structures for integrating services and processes in a SOA. The authors distinguish between two types of service-based processes, macro- and microflows. Macroflows describe long-running, higher-level processes and microflow short-running, rather technical processes. Based on this distinction, the authors identify several generic components of a process-driven SOA, such as a *macroflow engine* ([19], pp. 34–37) or a *configurable adapter repository* ([19], pp. 29/30). All in all, there are nine such generic structures of a process-oriented SOA.

3.2 Pattern Catalog Selection

A selection of pattern catalogs for the analysis should not be based on personal bias. Instead, pattern catalogs should be favored according to their importance for the domain in focus. A pattern catalog can be said to be important for a domain, if it contains patterns that help to satisfy domain-specific requirements. The more requirements a pattern catalog helps to satisfy, the more important it is considered here. [40] presents a comprehensive framework of requirements for B2Bi and a judgment of the importance of these requirements for the different layers of abstraction as described in section 2.1.1. This study can serve as a foundation for constructing a ranking of the pattern catalogs. A detailed discussion of the requirements is not possible here due to space limitations. In the following, relevant requirements will be mentioned shortly. For the complete description of a requirement, please refer to [40]. The next section describes the approach taken for constructing a ranking of pattern catalogs followed by the application of the approach and its result.

3.2.1 Approach

The approach for pattern catalog selection developed and applied here works in several steps. Although all pattern catalogs described in section 3.1 originate from the SOA and process language context, not all of them are a suitable basis for the analysis. In a first step, some catalogs will be excluded from further consideration for reasons related to their content. There is one primary reason for this step: All patterns considered in an analysis should be derived from real-world process models. If they are not, there is the risk that they do describe construct excesses [30]. There might be good reasons for the support for a pattern in a language from an engineering point of view, for example because it enables the achievement of a certain behavior in an elegant way. Yet, this does not mean the pattern should be directly available in a language. A language should only directly provide the structures that are frequently needed in real-world processes. If it provides any reasonable structure directly, it would simply get too complex and confusing. Therefore, pattern catalogs that can be used for pattern-based analysis should also base their patterns on empirical findings from real-world process models.

In a second step, each of the requirements from [40] is considered for each of the remaining pattern catalogs. A requirement is considered to be suitable as motivation for a pattern catalog if the catalog contains one or more patterns that help to fulfill the specification of the requirement. This results in a mapping from pattern catalogs to requirements. In [40], each requirement is also assessed according to its importance for one of the layers of abstraction as defined in section 2.1.1 ([40], pp. 39–42). This is done by assigning each of the combinations of abstraction layer and requirement a value, called *consider (csdr)* value ([40], p. 11). The possible range of values is % (not applicable), - (should not be considered), 0 (could be considered), 1 (should be considered) or 2 (strongly recommended to be considered)[8]. It can easily be seen that values of 0, 1 or 2 describe a positive relationship while values of % or - describe a negative relationship. So, if a pattern catalog can be related to multiple requirements

[8]The semantics are as follows: A csdr value of 0 for requirement x at abstraction layer y means that requirement x could be considered at layer y.

that have a positive relationship to the orchestration layer, the catalog is of importance to the orchestration layer[9].

Based on this result, in step three, a mapping from pattern catalogs to csdr values for the orchestration layer is constructed. This is done by reading the csdr values a requirement scored for the orchestration layer and counting the amount of values of all the requirements related to a catalog ([40], pp. 39–42). Pattern catalogs can then be directly compared based on their csdr scores and a pattern catalog that achieves higher values for the positive scores is considered to be more important for this analysis than a catalog with a smaller score. This constructs a ranking of the catalogs that determines in which order an analysis should proceed. The function for the comparison of the catalogs works as follows:

$$\begin{aligned} Catalog_i > Catalog_j \;\; \leftrightarrow \;\; & (csdr_2(i) > csdr_2(j)) \\ & \vee((csdr_2(i) = csdr_2(j)) \wedge (csdr_1(i) > csdr_1(j))) \\ & \vee((csdr_2(i) = csdr_2(j)) \wedge (csdr_1(i) = csdr_1(j)) \\ & \wedge(csdr_0(i) > csdr_0(j)) \end{aligned}$$

The function states that pattern catalog A is more important than catalog B if it scored higher in $csdr_2$ values than B ([40], pp. 39–42). If both are equal in their $csdr_2$ score, A is more important if it has a higher score of $csdr_1$ values. If the catalogs are also equal in their $csdr_1$ score, A is more important if it has a higher score in the $csdr_0$ values. If the pattern catalogs are equal even for these values, they are said to be of equal importance. In short, the approach of pattern catalog selection works as follows:

1. Exclude unsuitable catalogs.
2. Construct mapping: Pattern catalogs - requirements.
3. Determine amount of csdr values for each catalog.
4. Compute the ranking of the catalogs.

3.2.2 Evaluation and Results

In the first step, four pattern catalogs are excluded from further consideration. The nature of the patterns of these catalogs and the research method for deriving them suggested this decision. These are first the two integration pattern catalogs, the enterprise integration patterns [20] and the process-oriented integration patterns [19]. Second, the multi-party multi-message request-reply conversation patterns [28] and exception handling patterns [39]. The two integration pattern catalogs are excluded because they motivate their patterns from an engineering point of view and not through empirical findings in process models. They mostly concern how the architecture of a PAIS should be structured. Only a subset of the enterprise integration patterns can be considered to be directly relevant to process models. Many of these patterns concern the construction of a messaging-based architecture that is able to support the execution of processes in the first place. There is no reason why these processes in turn should support structures like a *Process Manager* ([20], pp. 312 - 321). Analogously, the process-based integration patterns

[9]The orchestration layer is of course identified as the combination of public and private orchestrations.

describe how to build an architecture that supports the execution of processes. They do not directly describe the structures that should be available in the processes that are then executed based on this architecture. The multi-party multi-message request-reply conversation patterns and the exception handling patterns are excluded for another reason. Although described as patterns by their authors and specifically aimed at the application in pattern-based analysis, they lack one of the basic properties of patterns. Patterns are recurring constructs based on empirical evidence for their importance. These pattern catalogs are not motivated by empirical evidence, but by considerations for a conceptual framework. In the case of the multi-party, multi-message, request-reply conversation patterns the aim is to characterize the structure of all possible multi-party multi-message request reply conversation scenarios. The exception handling patterns aim at characterizing all combinations of exception handling mechanisms that are possible in workflows. There is no reason why all possible conversation structures or all possible exception handling mechanisms should be of equal importance in realistic processes. Some conversation structures might be completely impractical for real processes and some exception handling mechanisms might be unreasonable from a realistic perspective. Which conversation structures and exception handling mechanisms are of direct relevance for real-world processes cannot be judged from the two studies and an empirical study is not provided for these pattern catalogs. Consequently there is no guarantee that these pattern catalogs do not measure construct excesses[10]. The remaining ten pattern catalogs now undergo the second step, the mapping of requirements to catalogs. For each catalog, the requirements it helps to fulfill are discussed shortly.

control-flow patterns: Obviously, the control-flow patterns provide *control-flow definition* (req 13 [40], p. 14). As some of the patterns, such as *Exclusive Choice*, allow for a data-based routing of the control-flow, also *data-oriented process definition* (req 15 [40], p. 15) can be fulfilled with this pattern catalog. The requirement *control flow patterns* (req 24 [40], p. 17) explicitly calls for the use of this pattern catalog. No other pattern catalog has a requirement solemnly devoted to it. The requirement *adaptability* (req 59 [40], p. 25) demands the full specification of the control-flow of process models (as opposed to underspecified process models). This requirement can also be fulfilled with the control-flow patterns. Analogously, *process flexibility by design* (req 60 [40], pp. 25/26) demands the full specification of the control-flow under exceptional circumstances. This of course can also be achieved with the help of control-flow patterns.
The *control-flow patterns* requirement makes it obvious that the control-flow patterns are of paramount importance in the B2Bi domain. For this study, this requirement serves as a wild card for the control-flow patterns catalog and leads to a slight deviation from the approach for pattern catalog selection presented before. Because they are explicitly called for in a requirement, the control-flow patterns automatically achieve the highest rank. Consequently, the analysis addresses this pattern catalog first. The other catalogs are still ranked and analyzed according to their csdr scores.

service interaction patterns: *Support for business transactions* (req 9 [40], p. 13) calls for the support for interactions between two partners. Such scenarios are captured by the service interaction patterns. The same applies to the *support for binary collaborations* (req 11 [40], p. 14). The interaction scenarios of the service interaction patterns of course

[10]Considering the fact that there are almost 3000 multi-party, multi-message, request-reply conversation patterns, there is a great likelihood that some patterns mark construct excesses

contain the *control-flow definition* (req 13 [40], p. 14) of these scenarios and some of them, such as routing scenarios, also contain a *data-oriented process definition* (req 15 [40], p. 15). The service interaction patterns do not only capture binary interactions, but also multi-party interactions and therefore assist in providing *support for multi-party collaborations* (req 21 [40], p. 16). In the interaction scenarios, the roles of the different partners are defined, thereby supporting *role modeling* (req 22 [40], pp. 16/17). The basic service interaction patterns describe the primitives for *asynchronous and synchronous interaction* (req 37 [40], p. 20). The more complex scenarios require *message correlation* (req 40 [40], p. 21) and help to define how this should be achieved for the scenarios at hand. By describing the interaction scenarios from the viewpoints of the different partners, the service interaction patterns allow for a *multi-level and multi-view description* (req 46 [40], p. 22) of the interaction scenarios. Finally, the best-case control-flow of the interaction scenarios is fully specified thereby supporting *adaptability* (req 59 [40], p. 25).

correlation patterns: In general, any asynchronous distributed interaction scenario requires message correlation. Therefore, the correlation patterns aid in providing *support for business transactions* (req 9 [40], p. 13), *support for binary collaborations* (req 11 [40], p. 14), *support for multi-party collaborations* (req 21 [40], p. 16), and obviously *message correlation* (req 40 [40], p. 21). *Integration partner binding* (req 17 [40], p. 15) calls for the identification of partners in a technical sense which can be supported by correlation mechanisms at run-time. Finally, message correlation is a prerequisite for a proper functioning of asynchronous interaction and therefore the correlation patterns also help to support *asynchronous and synchronous interaction* (req 37 [40], p. 20).

data patterns: The primary data transfered and processed in B2Bi processes are business documents. With the pattern category of data-based routing, the data patterns further provide support for *control flow definition* (req 13 [40], p. 14) and *data oriented process definition* (req 15 [40], p. 15). Four of the patterns of this category also concern *pre/post conditions of process/task executions* (req 16 [40], p. 15). B2Bi processes may require *metadata definition* (req 26 [40], p. 17) and need to communicate this data to a process instance. The category of external data interaction among other things concerns how such metadata can be passed into a process. The same argumentation applies to *configuration data for runtime systems* (req 44 [40], pp. 21/22). As pre- and post-conditions are an implicit realization of states, the data patterns also assist in supporting *state-based modeling* (req 49 [40], p. 23).

protocol patterns: The protocol patterns describe specific interaction scenarios. For these scenarios, they provide a *control flow definition* (req 13 [40], p. 14) and describe *pre/post conditions of process/task executions* (req 16 [40], p. 15). They further define roles and thereby help to support *role modeling* (req 22 [40], pp. 16/17). Again, the description of pre- and post-conditions also implies support for *state-based modeling* (req 49 [40], p. 23).

activity patterns: Just as the protocol patterns, the activity patterns describe common interaction scenarios. Therefore, the same requirements that are supported by the protocol patterns are also supported by the activity patterns. But in difference to the protocol patterns, the activity patterns also regard the style of interaction accomplished by the patterns and thereby aid in supporting *asynchronous and synchronous interaction* (req 37 [40], p. 20).

resource patterns: First of all, the resource patterns describe *pre/post conditions of process/-task executions* (req 16 [40], p. 15) and thus also help to support *state-based modeling* (req 49 [40], p. 23). *Role modeling* (req 22 [40], pp. 16/17) is directly captured in a specific pattern. The allocation of tasks to resources can be a way of *interfacing with backend systems* (req 45 [40], p. 22). *Process governance* (req 67 [40], p. 27) describes the enforcement of organizational policies which can be implemented using roles and rights management and is therefore also supported by this pattern catalog. As resources can be said to provide services, such as the ability to execute a certain task, the resource patterns aid in providing a *management of relationships among service/process providers and service/process users* (req 75 [40], pp. 29/30).

change patterns: The change patterns revolve around changes to the control-flow of a process and therefore also help to support *control flow definition* (req 13 [40], p. 14). They describe semantic constraints that must be fulfilled in spite of changes occurring to a process model. Therefore they assist in providing a *language for semantic constraint specification* (req 14 [40], p. 15). Change patterns help to ensure that *pre/post-conditions of process/task executions* remain valid in the presence of changes (req 16 [40], p. 15). Through this, change patterns aid in providing an *ease of maintenance* (req 36 [40], pp. 19/20) for process models. The focus of the change patterns is on the *consistency* (req 42 [40], p. 21) of the control-flow in spite of changes. By describing how parts of processes can be composed at runtime, the category of patterns for predefined changes essentially provides *flexibility by underspecification* (req 58 [40], p. 25). Obviously, *flexibility by change* (req 61 [40], p. 26) is also addressed by the change patterns. The surveillance of the compliance to semantic constraints provided by the change patterns provides a basic level of *validation* (req 65 [40], p. 27) for process models. As change is part of the lifecylce of processes, the change patterns also help to support *life-cycle management of B2Bi artifacts; methodology* (req 66 [40], p. 27). Finally, the support for changes is a way of ensuring *extensibility* (req 76 [40], p. 30) and *dynamism* (req 77 [40], p. 30) of process models.

time patterns: Time-constraints, formulated in the time patterns, are a basis for several *Quality of Service (QoS)* (req 38 [40], p. 20) features, which are an essential prerequisite for the *support for business transactions* (req 9 [40], p. 13). The time patterns concern time-based *control flow definition* (req 13 [40], p. 14). Just like business transactions, *asynchronous and synchronous interaction* (req 37 [40], p. 20) is not possible without the specification of timeouts and other time constraints. Finally, time constraints are organizational policies and therefore the time patterns also aid in supporting *process governance* (req 67 [40], p. 27).

process instantiation patterns: Several process instantiation patterns describe the *control-flow definition* (req 13 [40], p. 14) on process start-up. This especially concerns which elements of the process initially are enabled. Instance creation can vary based on input data, thereby supporting *data-oriented process definition* (req 15 [40], p. 15). The patterns essentially describe pre- and post-conditions of the instance creation and thus support *pre/post-conditions of process/task executions* (req 16 [40], p. 15) and consequently *state-based modeling* (req 49 [40], p. 23). The pattern categories of subscription and unsubscription deal with event-based instance creation which obviously is of assistance for supporting *event propagation* (req 28 [40], p. 18). The interaction style of the patterns of event-based categories is asynchronous. Consequently, the catalog also aids

Table 2: Ranking of Pattern Catalogs

Rank	Pattern Catalog	Requirements	$csdr_2$	$csdr_1$	$csdr_0$
1	Control-flow	13, 15, 24, 59, 60	7	3	0
2	Change	13, 14, 16, 36, 42, 58, 61, 65, 66, 76, 77	13	7	2
3	Service Interaction	9, 11, 13, 15, 21, 22, 37, 40, 46, 59	11	8	1
4	Time	9, 13, 37, 38, 67	10	0	0
5	Correlation	9, 11, 17, 20, 37, 40	8	4	0
6	Process Instantiation	13,15,16,28, 37, 49	5	5	2
7	Resource	16, 22, 45, 49, 67, 75	5	3	4
8	Data	13, 15, 16, 26, 44, 49	4	6	2
9	Activity	13, 16, 22, 37, 49	4	3	3
10	Protocol	13, 16, 22, 49	2	3	3

in supporting *asynchronous and synchronous interaction* (req 37 [40], p. 20).

Based on the previous discussion, now the csdr-scores can be calculated for each pattern catalog. Table 2 presents the resulting numbers. It also outlines the ranking of the catalogs which these numbers imply. The control-flow patterns should be analyzed first due to the wild card requirement. The change patterns achieved the highest csdr-scores.

3.3 Current State of Pattern-based Analysis

Table 3 summarizes the current state of evaluation of the two languages for the pattern catalogs that can be found in the ranking. Due to the popularity of pattern-based analysis and the large amount of languages available, various evaluations can be found. Nearly all of the pattern catalogs have been published in combination with an analysis of recent languages. WS-BPEL

Table 3: Current State of Evaluation of WS-BPEL and Windows Workflow

Pattern Catalog Name and Source	WS-BPEL	WF
Control-flow Patterns [38, 43]	1.1	3.5
Data Patterns [37]	1.1	-
Resource Patterns [35]	1.1	-
Change Patterns [51]	-	-
Time Patterns [23]	-	-
Protocol Patterns [44]	-	-
Activity Patterns [42]	-	-
Service Interaction Patterns [3]	1.1	-
Correlation Patterns [2]	2.0	-
Process Instantiation Patterns [10]	2.0	-

has been evaluated in [2,10,35,37–39]. Most of these studies evaluate version 1.1 of the language which was replaced in 2007 by version 2.0. As this version also introduces changes to the activities of WS-BPEL, such as the elimination of the `switch` activity and the introduction of the parallel `forEach` activity, it can be expected that also the degree of support provided by WS-BPEL has changed. Based on the pattern catalogs, there are several studies that perform additional analyses. [11, 52] compare the expressiveness of WS-BPEL to other Web Service composition languages. Both of the studies use several pattern catalogs.

So far, only one study dealing with the degree of support provided by WF can be found. Its support of control-flow patterns is examined in [55] and it is compared to WS-BPEL. Also this analysis can be considered to be outdated. It relies on the language version of WF incorporated in .NET 3.5. In .NET 4, WF was changed significantly. Some of these changes, like the extension of the BAL, can be expected to have thorough impact on the degree of support provided by WF. Yet most notably, the *state machine* workflow modeling style, in [55] a necessity for the support for a variety of patterns, has been removed. Instead the new flowchart modeling style has been introduced.

4 Streamlining Pattern-based Analysis

Today, the main problem with pattern-based analysis is a limited comparability of the results for different pattern catalogs and a limited selectivity of the support measure. The problem of comparability can be tackled by employing a unified notion for the validity of a candidate solution. The notion presented here is derived from the different methodologies used by the authors of the relevant pattern catalogs. The problem of selectivity is due to the fact that the traditional trivalent support measure is too coarse. This situation can be improved by using an alternative measure, the edit distance (cf. section 4.2). Altogether the approach presented here works in two steps:

1. For a given candidate solution, it is first determined whether it provides a *valid implementation* for a given pattern.
2. If so, the degree of support it provides is calculated. This calculation is done using an alternative scaling of the support measure, the edit distance based on high level change operations.

To demonstrate that the edit distance indeed provides a higher degree of selectivity than the traditional measure, the calculation of the degree of support is performed twice in the following analysis. Once, it is calculated using the edit distance and once using the traditional trivalent measure. For comparing the results for different pattern catalogs, it is necessary to unify the calculation not only of the edit distance, but also of the trivalent measure. The following sections outline the notion of implementation validity and the computation of the two support measures. A comprehensive example of the computation of the measures is provided in section 5.2, when the scope of the languages in focus has been defined.

4.1 Implementation Validity

The first step of the approach is to determine whether a given candidate solution is a valid implementation of a given pattern. Only a solution that fulfills this minimum criterion is able to provide support for a pattern. The decision whether this is the case is based on the structure and components of a pattern which are similar for all pattern catalogs at hand, although not all of the catalogs contain all of the aspects discussed below. Five components of a pattern are essential for determining the validity of an implementation.

Pattern description: The pattern description specifies the nature of the pattern and its *core aspects*. To provide a valid implementation, a candidate solution must cover all core aspects that are found in the pattern description, as explicitly stated in [3]. This minimum component can be found in any pattern catalog.

Pattern context: The *context*, in some cases called *issues* [42], describes several assumptions or criteria about the environment a pattern is operating in. To provide a valid implementation, at most one of these criteria may not be met by a candidate solution. This pays tribute to the fact that the support for a pattern should still be calculated even if minor aspects cannot be covered. These constraints can be inferred from the evaluation criteria of [35, 37, 38]. As an example, the *Structured Synchronizing Merge* pattern requires the

existence of a preceding *Multi-Choice* construct in a context criterion ([38], pp. 17–19). Context criteria can be found in [3, 23, 35, 37, 38, 42, 43].

Execution traces: Closely related to the pattern context is the notion of *execution traces* ([23], p. 98). An execution trace defines the structure of all possible execution sequences of activities that are valid for a given pattern. Examples are mathematical expressions, used in [2, 23, 50], or graphical notations such as Petri Nets, used in [38, 42]. If a formalization for execution traces is present, a candidate solution must also conform to these traces which is explicitly stated in [23, 50].

Design choices: In most cases, the definition of a pattern is flexible to some extent. Certain aspects are left open to the choice of the implementer of a pattern, which are described as *design choices* ([23], p. 97). Each design choice denotes a list of alternative aspects one of which can be chosen when implementing a pattern. A combination of different aspects from the design choices attached to a pattern then forms a *pattern variant* ([23], p. 97). A candidate solution must implement at least one pattern variant [23, 51], omitting at most one of the design choices of the variant. As an example, a solution for the *Durations* pattern that only supports maximum, but not minimum durations of activities still forms a valid implementation of the pattern ([23], p. 100). Design choices can be found in [3, 23, 42, 51].

Data types: A pattern might inherently depend on the availability of specific data types, such as dates or timestamps [2, 22, 23]. To provide support for a pattern, a corresponding data type must be available in a language. Additionally, if needed in a candidate solution, necessary operations for comparing or manipulating these types must be provided, as can be found in the evaluation contained in [22].

Figure 5 on the next page depicts the approach described here in the form of a binary decision tree.

For a candidate solution that provides a valid implementation, the degree of support can be calculated. The problem of qualifying the effort needed to implement a pattern can be interpreted as a question of *distance* between processes. Say process X is a process stub without specific functionality and process Y is an extension of X that adds exactly the solution of a pattern. The less distant X is to Y, the less effort is needed to transform X into Y. So, the support for a pattern in a language can also be measured by computing the distance between two processes written in the language, where one of the processes extends the other one with the implementation of a given pattern. Listing 2 outlines such process stubs for WF and WS-BPEL. The process stubs are semantically identical and consist of a receive activity embedded in a sequence activity, along with optional input parameters.

For WF, the process stub is not an ordinary workflow, but a workflow service. The `Receive` activity as well as the implementation of the pattern that follows are contained in a `Sequence` activity. The `Sequence` activity also defines a `Variable` that stores the input of the `Receive` activity. In WS-BPEL, the process stub contains necessary import definitions and the definition of one `partnerLink` which is inevitable for a working process. The control-flow of the minimum process is formed by a `receive` activity that creates a new process instance and uses a `variable` as input embedded in a `sequence` activity. The pattern implementation then succeeds the `receive` activity.

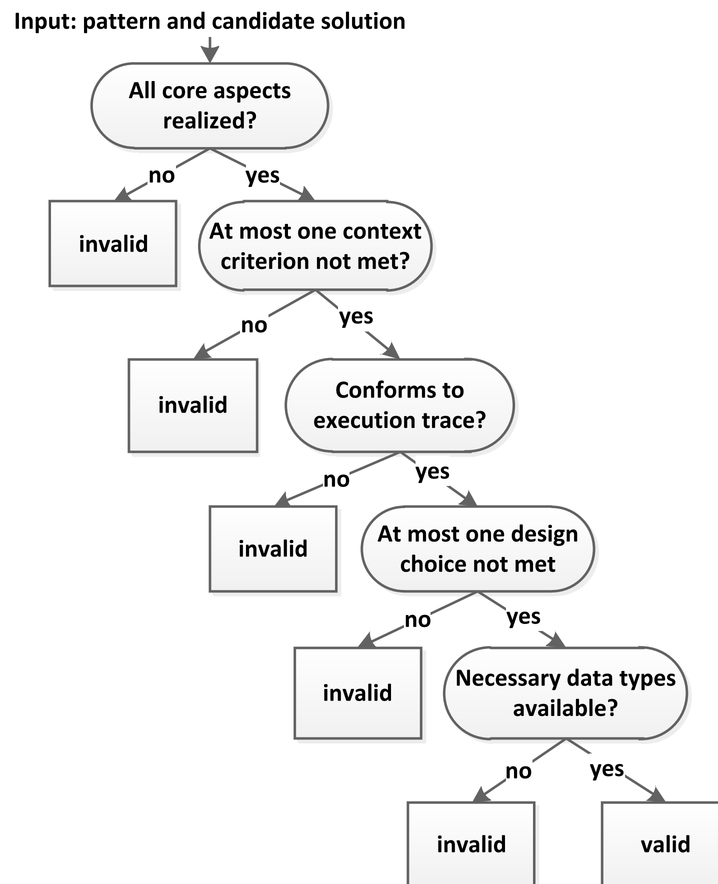


Figure 5: Decision Tree for Determining Implementation Validity

Listing 2: Process Stubs

```
1  <!--WF Process Stub-->
2  <WorkflowService>
3      <Sequence>
4          <Sequence.Variables>
5              <Variable Name="InstanceID" />
6          </Sequence.Variables>
7          <Receive CanCreateInstance="True" OperationName="StartProcess">
8              <ReceiveParametersContent>
9                  <OutArgument Key="Input">[InstanceID]</OutArgument>
10             </ReceiveParametersContent>
11         </Receive>
12         <!--Pattern Implementation-->
13  </Sequence>
14 </WorkflowService>
15
16 <!--WS-BPEL Process Stub-->
17 <process>
18    <import location="ProcessInterface.wsdl" />
19    <partnerLinks>
20        <partnerLink name="MyPartnerLink" myRole="patternRole" />
21    </partnerLinks>
22    <variables>
23        <variable name="StartProcessInput"/>
24    </variables>
25    <sequence>
```

```
26         <receive createInstance="yes" variable="StartProcessInput"
27          partnerLink="MyPartnerLink" operation="StartProcess"/>
28          <!--Pattern Implementation-->
29    </sequence>
30 </process>
```

Based on the validity of the solution and the process stubs, the degree of support provided by a solution can be calculated using different measures. The two measures employed here are the traditional trivalent measure and its improvement, the edit distance.

4.2 Edit Distance

[54] presents several measures for computing the similarity between process models. A foundation for these similarity measures that seems very applicable for the problem at hand is the *Levenshtein distance* [25], or *edit distance*. This distance measures the smallest distance between two strings by calculating the minimum number of change operations, being substitutions, insertions or deletions of characters that are needed to transform one string into another. For the problem at hand, the basis are of course process models and not strings. The models to be compared are a process stub, as demonstrated in listing 2 and a process extending this stub with the implementation of a pattern. Counting substitutions of characters would make no sense here, as the distance in concepts and constructs would get lost in syntactical noise. For example a language could tend to have higher distances simply because its activities have longer names. Much more applicable in this case are high level changes to the structure of the process model, as opposed to changes of characters. The difference is that high level changes comprise larger structures and satisfy minimalistic semantical constraints. Examples are the insertion of an activity and the setting of its name, the insertion of a variable and the setting of its name and type or the setting of a target variable and expression in an assignment. A concrete example for BPEL would be the configuration of correlation for a `receive` activity. This involves the creation of a `correlations` and a `correlation` element, the assignment of its name and potentially whether the correlation set should be initiated. Counting each syntactical modification, instead of the single high level operation *configure correlations*, adds noise to the final result. The intent of the edit distance here is after all not to capture differences in naming, but differences in concepts and constructs, because these differences better describe the effort needed by the user of a language. The edit distance can now be calculated by adding up the amount of high level insertions, substitutions and deletions needed. Using the same set of high level changes as basis for the edit distance in the assessment of different catalogs ensures comparability between the results. Generalizing the set of high level changes and making it applicable for different languages also provides comparability between the languages.

The edit distance discussed here also relates to the *graph-edit distance* [12,13]. This edit distance is used in [50] based on editor operations for demonstrating the necessity for the support for adaptation patterns. Obviously, this edit distance focuses on graph-oriented instead of block-structured process models. It computes the difference of process models based on substitutions, insertions, and deletions of nodes and edges in a process model ([13], pp. 6/7). This abstract set of operations is quite unspecific and does not take operations like the configuration of a node into account. In this study, a set of operations which is specific for orchestration languages and

is applicable for graph-oriented and block-structured process models is developed and applied.

The set of necessary operations was determined throughout the conduction of the analysis. Starting with a list of candidate operations, the relevance of an operation was determined by the fact that it could be applied in a large set of the process models developed here. In some cases, as especially for messaging activities, it became obvious that further change operations needed to be defined and the list was extended. This method resembles the method used by several authors for determining a set of patterns. The following list of operations is *complete* and *sufficient* concerning the developed process models. Completeness here means that no other change operations apart from the operations described were necessary. Sufficiency states that it was possible, using this set of operations, to calculate support values that provide a higher degree of selectivity than the traditional measure. Each of the following operations counts one point to the edit distance of a candidate solution.

Insert Activity: The insertion or substitution of an activity and the setting of the activity name. Any WS-BPEL activity and any WF activity counts to this. Further configuration of an inserted activity is not included. In a block-structured process model, inserting an activity also includes the change of the configuration of a composite activity. In such models, activities are necessarily positioned in the body of other activities and therefore the configuration of the composite activity is also included in the insertion. As an example, inserting an activity into the `Trigger` of a `PickBranch` activity in WF also configures the `PickBranch` activity.

Insert Edge: The insertion of an edge and the setting of its name. This operation is generally only available in a graph-oriented model. In WS-BPEL, edges are represented by `links` in a `flow` activity. In WF, 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: Apart from activities and edges, languages may use a variety of other 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 WS-BPEL, 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. All these configurations may obviously also include the definition of imports in the process model which are included in the insertion of the respective process element.

In WF, 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 and therefore are included in other edit operations. In WF, the insertion of a `Variable` involves the setting of its type and name. WF also allows to fix a default value for a variable on its definition which counts to the operation of inserting it. In WS-BPEL, this is not possible and must be achieved in a separate `assign` activity.

For the languages at hand, it would have been possible to split this operation into three distinct operations, *insert variable*, *insert partner reference*, and *insert correlations*. For generality and extensibility of the approach, the single operation is provided.

Change Configuration: The change of any default value of an activity to another value. In WS-BPEL or WF, 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 WS-BPEL where the configuration of the wait condition is fixed in a `for` or `until` element. Especially for messaging activities, there are several types of configurations that are independent of a specific language. Some of these configurations, necessary after the insertion of an activity, require the change of more than one attribute at a time, but are logically related. Therefore, these special configurations are captured in specific change operations. All other change operations in a process can be represented by the current operation.

Configure Messaging Properties: Messaging activities require the configuration of several properties, all related to the interface of the single operation 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 change operation. In WS-BPEL, the configuration of the messaging properties of an activity involves the setting of the `partnerLink`, `portType` and `operation`. In WF, it corresponds to the setting of the `ServiceContractName` and the `OperationName`.

Configure Addresses: 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 WS-BPEL, the current operation is not needed. Here the address of a service can already 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, as it does not make use of an explicitly 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, one of with is the creation of an `Endpoint` element for the messaging activity and the setting of its `AddressUri` and the `Binding` to be used.

Configure Correlations: Messaging activities might also require the configuration of message correlation. In general, this is a mapping from the parameters available to the activity to a predefined correlation set or variable. In WS-BPEL, 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, there are differences depending on whether a `CorrelationHandle` should be initialized or correlated on. In any case, 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 WS-BPEL, 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. In WF it implies the definition of a parameter and the mapping of the parameter to a predefined `Variable` using an expression.

Configure Expression: Several attributes may require expressions as values. Examples are logical expressions used in the termination condition of a loop. The construction of such an expression may require several steps and may involve the use of several operators in the expression language. As the parts of an expression are not useful in isolation, the construction of an expression counts as a single 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 allows for single assignment in an `Assign` activity, WS-BPEL allows for multiple ones using multiple `copy` elements.

Obviously, such change operations can be facilitated by using a sophisticated integrated development environment. The aim of this study however, is to measure the support provided by a language and not by tools available for the language. The edit distance as discussed here abstracts from the availability of specific tools that facilitate edit operations. The same applies to the representation of the language [21]. The edit operations are independent of whether the language is defined by a graphical or by a serialization format. This implies that the identification of constructs that add to the edit distance cannot easily be automated. Relying on the syntactical elements of a representation format such as XML tags or state machine nodes is not sufficient.

Finally, a solution to a pattern may not only comprise a single process, but multiple processes. This is the case for patterns that focus on interactions of processes, such as the service interaction pattern [3]. Here, a valid solution for a pattern requires at least two processes. The more complex multi-party and routing patterns even require a minimum of three processes. In this case, the edit distance can be computed for each of the processes separately. The sum of these edit distances then forms the edit distance for the overall solution.

4.3 Trivalent Measure

Providing a trivalent support measure that is applicable for different patterns is quite difficult. Making it applicable across the boundaries of pattern groups and catalogs is even harder. This is the case, because of the strongly varying nature and complexity of the different patterns. Aiming at comparability, it is very hard to qualify them according to only three dimensions. Unfortunately, it is only possible with several complex restrictions and exceptions which are outlined in the following. This discussion also emphasizes the advantage of the edit distance. There, a manageable set of edit operations is sufficient, instead of a variety of different, interdependent rules. The rules presented here aim at providing a trivalent measure that adheres to previous studies as far as possible. At the same time, it is supposed to maintain comparability between the analysis of different pattern catalogs. This however, is only possible to a limited extent, as can be seen in the following discussion. Table 4 summarizes the rules used to determine the support with the trivalent scaling.

Only essential constructs: Traditionally, in case a candidate solution is valid, the degree of support is mainly determined by the number of *constructs* necessary to implement the pattern. A candidate solution may provide full support for a pattern only if there is a single

construct that implements it. If a combination of two constructs is necessary, the solution still may provide partial support. If the number of constructs necessary exceeds two then the solution is not considered to provide support. Therefore, it is important to define what counts as a construct. In general, these are the activities used in a solution that are essential for a pattern. Essential activities relate to the core aspects of a pattern. Many preceding analyses base their assessment on the conceptual analysis of a language and not on executable processes. This is a contrast to the study at hand, where the support is calculated based on executable processes. An executable process normally requires the use of more constructs than those essential for a pattern, simply for providing the ability to execute it. For instance, such basic constructs are the sequences and receives that can be found in the process stubs presented in listing 2 on page 35. Obviously, non-essential constructs should not count to the trivalent measure.

Only activities: In conformance to previous analyses, only WS-BPEL and WF activities count as constructs [52, 55]. This means that all aspects necessary to produce a working solution that do not belong to the set of activities, do not influence the degree of support provided by the solution. In the preceding section, these aspects were denoted as auxiliary constructs. Any configuration of an activity that might be necessary to produce a working solution does also not influence the degree of support. Configurations are for example changes to the default values of the attributes of an activity or the specification of a boolean expression.

Exclude basic activities: As discussed, a process might require the use of several basic activities for achieving executability. Also counting such basic activities would render the trivalent measure meaningless. Most solutions would score the value of *no direct support*. To achieve meaningful values, it is necessary to exclude such basic activities from the calculation of the trivalent measure. Basic activities in general are sequence, assign or delay activities. Unless these activities are essential to a pattern, they do not count to the trivalent measure. Unfortunately, the notion of what counts as a basic activity needs to be adjusted for several pattern catalogs or categories. This is the case, because several pattern catalogs describe much more complex structures than others. Applying the same notion of basic activities, renders the trivalent measure meaningless for a set of these catalogs. For instance, the categories two to four of the service interaction patterns [3], primitive messaging operations such as “sends” or “receives”, do count as basic operations and thus do not influence the trivalent measure. Omitting this restriction, an evaluation would present a value of *no direct support* for all patterns for both languages in focus here and this is obviously not desirable. If such restrictions apply, they will be stated per pattern or pattern catalog during the analysis.

Composite constructs: In accordance to previous analyses [52, 55], there is one more general restriction on what counts to the amount of constructs. Some activities are an integral part of other activities. Such integral parts do not increase the amount of constructs needed in a solution. For example, a `pick` activity in WS-BPEL must always contain at least one `onMessage` activity. Similarly, a `Pick` activity in WF must always contain at least one `PickBranch` activity. A `PickBranch` activity also must contain a `Trigger` activity which does not increase the amount of constructs it represents. A combination of such activities does only count as one construct, although strictly speaking it consists of multiple activities.

Table 4: Computation of the Trivalent Support Measure

Degree of support	General Criteria
Direct support (+)	All context criteria fulfilled All design choices met At most one essential construct
Partial support (+/-)	At most one context criterion not met At most one design choice not met At most two essential constructs
No direct support (-)	Solution is not valid More than two essential constructs

Increased number of constructs: Also, some patterns require the existence of at least two or more constructs instead of just one. An example is the Critical Section pattern ([38], pp. 72/73), where two constructs are executed with mutual exclusion. In such a case, these two constructs do only count as a single construct. Otherwise, a rating of direct support would not be achievable.

Context criteria and design choices: Apart from the amount of constructs, two more restrictions influence the computation of the trivalent measure. In accordance to the evaluation criteria of several studies [35, 37, 38], the support for context criteria and design choices also influences the trivalent measure. To provide direct support, a solution must fulfill all context criteria and design choices. If at most one context criterion and one design choice cannot be met, a solution may at most provide partial support. In all other cases, the solution does not form a valid implementation and is insufficient to provide support, independent of the amount of constructs used.

If more than one process is required for a valid solution to a pattern, the support according to the trivalent measure can be calculated for each of the processes separately. The minimum degree provided by the processes forms the overall degree of support provided by the pattern. Adding up the constructs used by all processes would again produce meaningless values for the overall support.

5 Analysis of Pattern Support

The following sections document the application of the approach presented in the previous section. The languages WS-BPEL, Sun BPEL (cf. section 5.1.2), and WF are assessed using the four most important pattern catalogs as determined in section 3. These are the control-flow, the change, the service interaction, and the time patterns.

As the realization of one pattern variant is sufficient to compute the support provided by a language, one solution is provided for each pattern for each of the languages. Based on this solution, the two support measures are calculated. It is possible for the two measures to conflict with each other, even though this is rarely the case. For example, a solution that provides *no direct support* concerning the trivalent measure might achieve a smaller value for the edit distance than a solution that provides *direct* support. This can happen, because the latter solution requires an extensive configuration of the activity used. The processes developed here aim at minimizing the edit distance. So, if there is a conflict, the solution with the smaller edit distance is presented. That way, the processes establish an upper bound for the edit complexity of the realization of a pattern in the languages. It might still be possible that more effective solutions with a smaller distance value can be constructed. This is especially true for the very complex solutions that incorporate a high degree of freedom. Still, the processes demonstrate that solutions are realizable that require at most a certain amount of operations.

In the following, a discussion on the precise scope of the two languages and the nature of the process models developed precedes the analysis, along with an example calculation of the support measures[11]. During the discussion of the results of the analysis, code fragments are provided for comprehensibility. These code fragments are extracted from the process models that implement a pattern. It is important to note that these fragments do only outline the functioning of a solution. They abstract from details as far as possible and are therefore not sufficient for computing the support measures. To compute the support measures correctly, it is necessary to look at the underlying process models directly[12]. For comprehensibility, the implications of the results for each of the pattern catalogs are discussed directly after its analysis. This discussion is summarized and condensed at the end of the chapter.

5.1 Language Scope and Logging

Apart from being the basis for the computation of the support measures, the processes developed here also serve another main purpose. An executable process is an undeniable demonstration that a certain feature of a language works as expected. To be able to verify that a certain feature works as expected and that a given implementation adheres to the execution trace of a pattern, a process must provide output during its execution. Here, the output takes the form of log messages produced by logging activities that are executed at significant points in a process. In most cases, these activities have no impact on the validity of a solution and need to be

[11] All process models that are described here can be downloaded at `http://www.uni-bamberg.de/en/pi/bereich/research/software-projects/pattern-based-analysis-of-orchestration-languages/`

[12] Strictly speaking, this applies mostly to the edit distance. The trivalent measure can in many cases be correctly inferred from the code fragments, but this is not guaranteed to be true in all cases.

excluded when computing the support measures. In some cases, the need for logging activities results in a need for other composite activities. As an example, consider the case where, for a valid solution to a certain pattern, it would have been sufficient to use a single WS-BPEL `invoke` activity as the child of an `onMessage` activity. The `onMessage` activity accepts only a single activity as child. To provide output for the sending of the message, it is necessary to wrap the `invoke` activity in a `sequence` activity and insert a logging activity after it. In such cases, the composite activity is also not relevant for the validity of a solution and does not influence the degree of support. Finally, in some cases, activities require the existence of arbitrary other activities for syntactical reasons. An example is again an `onMessage` activity that requires the existence of a child activity. To fulfill this requirement, it is sufficient to place a logging activity in the `onMessage` activity. In this case, the logging activity does influence the validity of the solution and therefore is considered when computing the degree of support. For the trivalent scaling, it is counted as a basic activity. For the edit distance, it does count as a normal activity. The configuration of the activity is not included, as this does no longer have an impact on the validity of a solution.

5.1.1 WF Scope and Environment

For WF, the evaluation of the pattern support is focused on the BAL and the parts of the .NET class library that are needed by BAL activities. As discussed in section 2.3.1, workflows can either be developed in code using one of the general purpose languages of .NET or declaratively using XAML. Workflows that are implemented in code can also leverage any of the concepts and structures provided by the general purpose language in which they are written. This way, any of the patterns can be implemented as a new activity in code. However, the aim here is not to assess the support provided by one of the .NET general purpose languages, but the built-in support provided by WF. Solutions in XAML are limited to these built-in aspects. That is why the WF processes are developed in XAML. Solutions for patterns based on code are out of scope. Only combinations and configurations of, and functions provided through the activities in the BAL and selected classes of the .NET class library may serve as solutions for a pattern. Also, WF comes in a procedural and a flowchart modeling style that can be mixed when implementing a workflow. In principle, solutions using any of these styles or a combination of both would be sufficient to provide support for a pattern, as long as only built-in activities are used. For the analysis, a pattern is considered to be supported if it is supported by a solution in at least one of the modeling styles. It needs not to be supported by both of them. In general, solutions are developed using the procedural modeling style. They generally have a smaller distance value which is partly due to the process stub as can be found in listing 2 on page 35. Still, in some cases valid solutions can only be implemented using the flowchart modeling style. These assumptions for the analysis of WF 4 correspond to the assumptions made in [55] for the analysis of WF 3.5. The Integrated Development Environment (IDE) used to develop the workflows is *Visual Studio 2010 Ultimate*. This is the most recent environment for developing .NET based applications at the time of writing[13].

The logging activity in WF is a self-developed C# code activity named `LogCodeActivity`. The fact that it is developed in code is no problem as it does not have impact on the validity of a

[13]Installation instructions and references are provided in the appendix in section B.1.

solution. This activity writes a message to the *Windows Event Log* which provides a sophisticated logging database[14]. Listing 3 demonstrates an exemplary use of a `LogCodeActivity` in a workflow.

Listing 3: Exemplary Use of a `LogCodeActivity`

```
1 <!--Other activities-->
2 <LogCodeActivity LogName="NameOfThePattern" Message="Process is in state xyz" MessageNumber
    ="3" />
3 <!--Other activities-->
```

5.1.2 WS-BPEL Scope and Environment

For WS-BPEL, the evaluation is focused on the activities defined in the standard document [31] and the standards used by these activities. These are basically XPath 1.0 as expression language, using XSD basic data types, and WSDL 1.1. The definition of the WSDL files that are needed by the processes is out of scope and does not influence the degree of support provided by the WS-BPEL solutions. Strictly speaking, WS-BPEL allows for the use of any expression language ([31], p. 49). However, the standard only requires the support for `XPath` 1.0 in an implementation. A WS-BPEL process that uses another expression language can no longer be ported between different engines, although it conforms to the standard. Therefore in this analysis, only XPath 1.0 is considered. As WS-BPEL is a standard, its evaluation can only be performed in theory. To provide executable processes in the language, an implementation of the standard needs to be used. Any implementation however may deviate from the standard and also introduce proprietary extensions ([31], pp. 164–166). Therefore, the pattern support of an implementation of WS-BPEL can differ from the WS-BPEL standard and needs to be treated as a separate language. This distinction conforms to previous evaluations of WS-BPEL, like [38] where WS-BPEL 1.1 and its implementation Oracle BPEL were in focus. The WS-BPEL implementation used here is the *Open ESB BPEL Service Engine* provided by Sun Microsystems, for short Sun BPEL, as part of the OpenESB project[15]. This is an open source Enterprise Service Bus (ESB) that, among other features, provides an engine for executing WS-BPEL processes[16]. The OpenESB distribution used in the study is the *GlassFishESB* which combines the OpenESB, the *GlassFish* application server, and the *Netbeans* IDE. The GlassFishESB 2.2 was the most recent distribution at the time of writing, incorporating the Netbeans IDE 6.7.1 and GlassFish application server 2.1.1. The pattern support of Sun BPEL is determined by its degree of support for the standard activities and the extensions it provides. Sun BPEL also provides a mechanism for executing Java code in a WS-BPEL process. For this feature, the same considerations as for code activities in WF apply. Support for a pattern may not be achieved by embedding Java code in a process. Executable processes can of course only be provided in case a pattern is supported by Sun BPEL. For WS-BPEL, the validity of an

[14] For details on the logging mechanism and how to view the logs, please refer to sections B.4 and B.5 in the appendix.

[15] Due to the restructuring of the resources of Sun Microsystems after its acquisition by Oracle, many project sites are being moved. At the time of writing, a working link to the project page was `http://wiki.open-esb.java.net/`.

[16] The current degree of support for WS-BPEL 2.0 features by Sun BPEL is documented at `http://wiki.open-esb.java.net/Wiki.jsp?page=BPELDesignerAndServiceEngineFeatures`.

implementation needs to be inferred from the description of the activities in the standard. Still, code fragments and sample processes can be provided in case a pattern is only supported by WS-BPEL, but not by Sun BPEL. In general, the discussion and code fragments are provided for both WS-BPEL and Sun BPEL in combination, referencing only WS-BPEL. Only in the case where its degree of support differs from WS-BPEL, Sun BPEL is discussed separately.

To trace the execution of a Sun BPEL process, its logging capability is used. One of the extensions to WS-BPEL that Sun BPEL provides are trace extensions. These extensions can be used in any activity and allow to log the contents of a `variable` at the start or end of the execution of the activity. Although trace extensions can be attached to any activity, here `assign` activities are used as logging activities. An example of a Sun BPEL logging activity can be found in listing 4. As for WF, logging activities do not influence the support measures, because they only exist to verify that a process really executes as intended. In Sun BPEL, also variables were defined in the process model to enable logging. If the sole purpose of a variable is logging[17], it does not influence the support measures. In some cases, variables that are needed as input or output in messaging activities, could also be alienated for logging purposes. If so, these variables do count normally. If an `assign` activity only modifies a logging `variable`, it does count as a pure logging activity and consequently is not counted into the support measures. If it also modifies variables used in messaging or control-flow activities, it does influence the support measures as defined in the previous section.

Listing 4: Sun BPEL Trace Extension

```
1 <assign name="LogActivity">
2     <trace>
3      <log level="info" location="onComplete">
4           <from variable="logMessage"/>
5      </log>
6     </trace>
7     <!--copy a meaningful message to variable logMessage-->
8 </assign>
```

5.2 Example Calculation

The last sections presented several rules and restrictions on the calculation of the support measures which makes their computation non-trivial. Therefore, this section outlines their calculation in an exemplary manner. The following code fragments in WS-BPEL and WF are almost complete compared to an executable process. Just as in other parts of the study, XML namespaces are left out for the sake of readability. Furthermore, XAML contains a variety of attributes and elements only relating to the formatting of the activities in the XAML designer of Visual Studio. Such elements are contained in the process models, but are left out here. The following processes are the detailed solutions to the Milestone pattern. For a description of the pattern and a discussion on these solutions, please refer to page 69. For this pattern it is the case that neither the trivalent measure, nor the edit distance discriminate. Also, the general structure of the two processes is very similar. Both solutions provide an edit distance of eleven and partial support. At least, the change operations that comprise the edit distance illustrate several differences in the implementations.

[17]This is indicated, by the fact that the variable is of name `logMessage` and type `LogMessage`.

Listing 5: Calculation Example for WF

```
<WorkflowService Name="Milestone">
  <Sequence>
    <Sequence.Variables>
      <Variable TypeArguments="Int32" Name="InstanceID" />
      <Variable TypeArguments="CorrelationHandle" Name="CorrelationHandle" />
    </Sequence.Variables>
    <Receive CanCreateInstance="True" OperationName="StartProcess" ServiceContractName="
        Milestone">
      <Receive.CorrelationInitializers>
        <QueryCorrelationInitializer CorrelationHandle="[CorrelationHandle]">
          <XPathMessageQuery Key="key1">
            <XPathMessageQuery.Namespaces>
              <XPathMessageContextMarkup>
                <String Key="xgSc">http://tempuri.org/</String>
              </XPathMessageContextMarkup>
            </XPathMessageQuery.Namespaces>body()/StartProcess/Input</XPathMessageQuery>
        </QueryCorrelationInitializer>
      </Receive.CorrelationInitializers>
      <ReceiveParametersContent>
        <OutArgument TypeArguments="Int32" Key="Input">[InstanceID]</p:OutArgument>
      </ReceiveParametersContent>
    </Receive>
    <LogCodeActivity LogName="Milestone" Message="Entering milestone state. Milestone
        activity executable for 5 seconds" MessageNumber="1" />
    <Pick>
      <PickBranch DisplayName="Branch1">
        <PickBranch.Trigger>
          <Delay Duration="[TimeSpan.FromSeconds(5)]" />
        </PickBranch.Trigger>
      </PickBranch>
      <PickBranch DisplayName="Branch2">
        <PickBranch.Trigger>
          <Receive CorrelatesWith="[CorrelationHandle]" OperationName="
              ExecuteMilestoneActivity" ServiceContractName="Milestone">
            <Receive.CorrelatesOn>
              <XPathMessageQuery Key="key1">
                <XPathMessageQuery.Namespaces>
                  <XPathMessageContextMarkup>
                    <String Key="xgSc">http://tempuri.org/</String>
                  </XPathMessageContextMarkup>
                </XPathMessageQuery.Namespaces>body()
                    /ExecuteMilestoneActivity/Input</XPathMessageQuery>
            </Receive.CorrelatesOn>
            <ReceiveParametersContent>
              <OutArgument TypeArguments="Int32" Key="Input" />
            </ReceiveParametersContent>
          </Receive>
        </PickBranch.Trigger>
        <LogCodeActivity LogName="Milestone" Message="Executing optional milestone activity
             and exiting milestone state" MessageNumber="2" />
      </PickBranch>
    </Pick>
    <LogCodeActivity LogName="Milestone" Message="Exited milestone state" MessageNumber="3"
         />
  </Sequence>
</WorkflowService>
```

The edit distance of the WF solution comprises the following sequence of change operations:

1. **Insert auxiliary construct**: create `Variable` with name `CorrelationHandle` in the scope of the `Sequence` and set its type (Line 5).

2. **Configure correlations**: create `CorrelationInitializers` for `Receive` and configure the `XPathMessageQuery` (Lines 8 - 17).
3. **Insert activity**: create `Pick` activity (Line 23).
4. **Insert activity**: create `PickBranch` activity (Line 24).
5. **Insert activity**: insert `Delay` activity in the `Trigger` of the `PickBranch` (Line 26).

6. **Configure expression**: set `Duration` attribute of `Delay` activity to a `TimeSpan` expression (Line 26).
7. **Insert activity**: create `PickBranch` activity (Line 29).
8. **Insert activity**: insert `Receive` activity in the `Trigger` of the `PickBranch` (Line 31).
9. **Configure messaging properties**: set `OperationName` and `ServiceContractName` of the `Receive` activity (Line 31).
10. **Configure parameters**: create `OutArgument` for the `Receive` activity (Lines 40 - 42).
11. **Configure correlations**: set `CorrelatesWith` attribute of the `Receive` activity (Line 31) and configure the `XPathMessageQuery` (Lines 32 - 39).

As they do not influence the validity of the solution, none of the `LogCodeActivities` (Lines 22, 45, 48) counts to the edit distance.

The following considerations constitute the trivalent support measure: The insertion of the variable in operation 1, as well as all configuration operations, do not influence the traditional support measure. Of course, also the `Sequence` and the initial `Receive` that form the process stub do not count. The `Pick` and the first `PickBranch` inserted in operations 3 and 4 count as a single construct, as the `PickBranch` is an integral part of the `Pick`. Also the `Delay` activity inserted in operation 5 does not increase the amount of constructs. It is in turn an integral part of the first `PickBranch`, because this `PickBranch` requires a `Trigger`. The second `PickBranch` inserted in operation 7, including the `Receive` activity in its `Trigger` inserted in operation 8 does count. Together, this amounts to two constructs, the `Pick` and the second `PickBranch`. One pattern variant (deadline) of the milestone is fully implemented and no context criteria are violated. All in all, this sums up to a degree of partial support due to the amount of constructs needed.

Listing 6: Calculation Example for WS-BPEL

```
1  <process name="Milestone">
2    <import location="../../Pattern.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
3    <import location="../../Logging.xsd" importType="http://www.w3.org/2001/XMLSchema"/>
4    <import location="../../Properties.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
5    <partnerLinks>
6      <partnerLink name="PatternLink" partnerLinkType="PatternPartnerLinkType" myRole="
           patternRole"/>
7    </partnerLinks>
8    <variables>
9      <variable name="logMessage" type="LogMessage"/>
10   <variable name="StartProcessInstanceInput" messageType="
         startProcessInstanceRequestMessage"/>
11   </variables>
12   <correlationSets>
13     <correlationSet name="CorrelationSet" properties="correlationId"/>
14   </correlationSets>
15
16   <sequence>
```

```
17     <receive name="Receive" partnerLink="PatternLink" operation="startProcessInstanceAsync"
            portType="PatternPortType" createInstance="yes" variable="
            StartProcessInstanceInput">
18       <correlations>
19         <correlation set="CorrelationSet" initiate="yes"/>
20       </correlations>
21     </receive>
22     <assign name="Assign0">
23         <trace>
24             <log level="info" location="onComplete">
25             <from variable="logMessage"/>
26         </log>
27         </trace>
28       <copy>
29         <from variable="StartProcessInstanceInput" part="inputPart"/>
30         <to>$logMessage/log:processId</to>
31       </copy>
32       <copy>
33         <from>"Milestone"</from>
34         <to>$logMessage/pattern</to>
35       </copy>
36       <copy>
37         <from>"Entering milestone state. Milestone activity executable for five seconds"
             </from>
38         <to>$logMessage/message</to>
39       </copy>
40     </assign>
41     <pick name="Pick" createInstance="no">
42       <onMessage partnerLink="PatternLink" operation="sendAsyncMessage" portType="
           PatternPortType" variable="StartProcessInstanceInput">
43         <correlations>
44             <correlation set="CorrelationSet" initiate="no"/>
45         </correlations>
46         <assign name="Assign1">
47             <trace>
48             <log level="info" location="onComplete">
49                 <from variable="logMessage"/>
50             </log>
51             </trace>
52           <copy>
53             <from>"Executing milestone activity. Leaving milestone state"</from>
54             <to>$logMessage/message</to>
55           </copy>
56         </assign>
57       </onMessage>
58       <onAlarm>
59         <for>"P0Y0M0DT0H0M5.0S"</for>
60             <assign name="Assign2">
61             <trace>
62                 <log level="info" location="onComplete">
63                     <from variable="logMessage"/>
64               </log>
65             </trace>
66             <copy>
67                 <from>"Deadline expired. Leaving milestone state"</from>
68                 <to>$logMessage/message</to>
69             </copy>
70             </assign>
71         </onAlarm>
72      </pick>
73     </sequence>
74 </process>
```

The edit distance of the WS-BPEL solution comprises the following sequence of change operations:

1. **Insert auxiliary construct**: create `correlationSet` and set its name and properties (Lines 12 - 14).
2. **Configure correlations**: create `correlations` for `receive` activity and set its `initiate` attribute to `yes` (Lines 18 - 20).
3. **Insert activity**: create `pick` activity (Line 41).
4. **Insert activity**: create `onMessage` activity (Line 42).
5. **Configure messaging properties**: set `partnerLink`, `operation` and `portType` of `onMessage` (Line 42).
6. **Configure parameters**: set `variable` of `onMessage` (Line 42).
7. **Configure correlations**: create `correlations` for `onMessage` (Lines 43 - 45).
8. **Insert activity**: insert new `assign` in `onMessage` (Line 46).
9. **Insert activity**: create `onAlarm` (Line 58).
10. **Change configuration**: set the duration string in the `for` element of the `onAlarm` (Line 59).
11. **Insert activity**: insert new `assign` in `onAlarm` (Line 60).

The additional variable `logMessage` (Line 9) does not count to the edit distance, as it only relates to logging. The `assign` activities are logging activities and their configuration does not count to the edit distance. However the insertion of two of them is needed, as otherwise the `onAlarm` and `onMessage` would be syntactically incorrect. Therefore, their insertion counts to the edit distance.

The trivalent support measure is made up of the following considerations: The `correlationSet` inserted with operation 1 does not count to the trivalent measure. The `pick` and `onMessage` activities inserted in operations 3 and 4 count as a single construct, due to the same reasons as for the `Pick` and `PickBranch` activities in WF. The `onAlarm` activity, on the other hand counts as construct. The first `assign` activity is a pure logging activity and does not count as construct. The following two `assign` activities do count as basic activities as defined in section 4.3 and therefore do also not add to the amount of constructs. All in all, two constructs are used, the `pick` and the `onAlarm`. As a pattern variant is implemented without the violation of context criteria, partial support is provided.

5.3 Control-flow Patterns

5.3.1 Analysis of Control-flow Patterns

The sources of the control-flow patterns are [38, 43]. Here, primarily [38] is used. The patterns have been revised and benefited from a more precise description, augmentation, and better formalization with Petri Nets in this version. It is important to note that the definitions of some patterns, such as the Arbitrary Cycles pattern, have changed from [43] to [38]. In combination with [52], these sources also present an evalution of WS-BPEL 1.1 and discuss how the patterns could be implemented. As WS-BPEL 1.1 is similar to WS-BPEL 2.0, also the solutions to the patterns are. Consequently, the solutions discussed here are based on those presented for WS-BPEL 1.1 in [38, 43, 52]. In some cases, as for the basic patterns, the solutions are practically identical. In other cases, as for the Arbitrary Cycles pattern or the Discriminator patterns, the evaluation here differs from the previous studies which did not consider these patterns to be supported in WS-BPEL 1.1. In such cases, also the solutions presented here are new. Also for WF, an analysis of a preceding language version, WF 3.5, can be found in [55]. Therefore, also in this case the solutions presented here are based on those discussed in this study. However, WF has undergone much more significant changes from version 3.5 to 4.0. Consequently, also the solutions presented here differ to a larger extent from those presented earlier than the solutions for WS-BPEL. For a description of the solutions in WS-BPEL 1.1 and WF 3.5, please refer to the respective sources [38, 43, 52, 55]. The discussion of the different patterns is sorted according to their classification. This is not necessarily the order in which they appear in the publication.

Basic Control-flow Patterns:

WCP-1 Sequence: It is possible to define a sequential precedence relationship between activities ([38], pp. 8/9).
WF: WF supports this pattern directly with the `Sequence` activity. This activity is available in the process stub without modifying it. Only two activities need to be inserted. Otherwise, no precedence relationship would be present due to a lack of activities. So, the edit distance of the solution is two.
WS-BPEL: Also WS-BPEL offers support with the `sequence` activity, providing identical support measures.

Listing 7: Sequence Pattern

```
1 <!--WF Solution-->
2 <Sequence>
3     <!--Activity 1-->
4     <!--Activity 2-->
5 </Sequence>
6
7 <!--BPEL Solution-->
8 <sequence>
9     <!--Activity 1-->
10    <!--Activity 2-->
11 </sequence>
```

WCP-2 Parallel Split: The control-flow is divided into two or more concurrent branches ([38], pp. 9/10).
WF: In WF, this behavior can be accomplished using the `Parallel` activity. This activity splits the control-flow into several concurrent branches. However, the activities contained in it are not necessarily executed concurrently. The `Parallel` activity schedules the activities it contains for immediate execution, as opposed to the `Sequence` activity, which schedules them for execution one after the other. So, the activities in the `Parallel` could be executed at the same time. As WF has a single-threaded execution model, the `Parallel` activity produces an interleaving of the contained activities, but not a true concurrent execution. This is still valid concerning the pattern definition. It is demanded that the control-flow is split into two concurrent branches, these branches must not necessarily execute at the same time. True parallelism can still be achieved, but it depends on the activities embedded in the `Parallel` activity. Any activity whose type is derived from `AsyncCodeActivity` executes on a separate thread and thus concurrently. The edit distance of the solution is three, because at least two child activities are needed, without which there would be no split of the control-flow.
WS-BPEL: This pattern is directly supported using the `flow` activity, which executes its children concurrently. The edit distance of this solution is also three. As for the implementation of the `flow` activity in Sun BPEL, similar limitations as for WF apply. Again, the activities start their execution in the lexicographical order in which they are defined.

Listing 8: Parallel Split and Synchronization Pattern

```
1  <!--WF Solution-->
2  <Parallel>
3      <!--First Branch-->
4      <!--Second Branch-->
5  </Parallel>
6
7  <!--BPEL Solution-->
8  <flow>
9      <!--First Branch-->
10     <!--Second Branch-->
11 </flow>
```

WCP-3 Synchronization: The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have completed ([38], pp. 10/11).
WF: The `Parallel` activity automatically synchronizes all contained branches after they have executed and triggers subsequent activities. The solution to this pattern is identical to the solution for the Parallel Split. The edit distance is therefore also three.
WS-BPEL: As for WF, also for WS-BPEL the `flow` provides automatic synchronization for its contained activities and the solution is also identical to the one for the Parallel Split pattern.

WCP-4 Exclusive Choice: The divergence of the control-flow into two or more branches exactly one of which is enabled based on the evaluation of a logical expression ([38], pp. 12/13).
WF: This pattern can be supported by the `If` or the generic `Switch<T>` activity. While the `If` activity allows for only two branches, the `Switch<T>` allows for more, depending on the amount of values of the type for which the activity is instantiated. A minimal solution with the `If` activity and two alternatives has an edit distance of four and provides direct support.
WS-BPEL: Direct support is provided with the `if` activity in combination with its child elements, `ifElse` and `else`. An equally minimal solution has the same edit distance.

Listing 9: Exclusive Choice and Simple Merge Pattern

```
<!--WF Solution-->
<If Condition="BooleanExpression">
    <If.Then>
        <!--First Alternative-->
    </If.Then>
    <If.Else>
        <!--Second Alternative-->
    </If.Else>
</if>

<!--BPEL Solution-->
<if>
    <condition>"BooleanExpression"</condition>
    <!--First Alternative-->
    <else>
        <!--Second Alternative-->
    </else>
</if>
```

WCP-5 Simple Merge: The convergence of two or more alternative branches into a single branch, which is enabled each time one of the alternative branches completes ([38], pp. 13/14).
Solutions: The solutions to the previous pattern automatically merge the diverged branches into a single flow of control. The support measures for this pattern are therefore identical to the previous one.

Advanced Branching and Synchronization Patterns:

WCP-6 Multi-Choice: The divergence of the control-flow into two or more branches multiple of which are enabled based on the evaluation of distinct logical expressions. This is an extension of the Exclusive Choice pattern, where multiple activations are possible ([38], pp. 15/16).
WF: There is no single BAL activity that is able to satisfy the core aspects of this pattern. Neither the `Switch<T>`, nor the `FlowSwitch<T>` allow for distinct conditions of their outgoing links or multiple activations. Instead, always only exactly one subsequent branch can be enabled. The `ConditionedActivityGroup` activity, that supported this pattern in WF 3.5 [55], does not longer exist and was not replaced by a corresponding construct. Nevertheless, there is a workaround solution that is applicable in practically any language and consists of a combination of XOR- and AND-Splits. In WF, it can be accomplished by a `Parallel` activity that contains multiple `If` activities which possibly enable several branches. This solution does provide partial support (as a minimum of two branches are needed in a working solution, only one of them adds to the amount of constructs) and its edit distance of seven shows that it is not so complicated to implement.
WS-BPEL: WS-BPEL provides a single construct for the Multi Choice. This is the `flow` activity in combination with `links`. `Links` allow to specify conditions under which their target activity is executed. These conditions can be distinct. By using an activity in a `flow` activity as the source of multiple `links`, the Multi-Choice pattern can be implemented. This solution provides direct support and its edit distance amounts to eight. Sun BPEL however, does not support `links` in the `flow` activity. Consequently, a solution similar to the one in WF has to be used. This solution is also slightly cheaper. Again, it does provide partial support and its edit distance amounts to seven.

Listing 10: Multi-Choice and Structured Synchronizing Merge Patterns

```
<!--WF Solution-->
<Parallel>
    <If Condition="BooleanExpression1">
        <If.Then>
            <!--First Branch-->
        </If.Then>
    </If>
    <If Condition="BooleanExpression2">
        <If.Then>
            <!--Second Branch-->
        </If.Then>
    </If>
</Parallel>

<!--One-construct BPEL Solution-->
<flow>
    <links>
        <link name="Activity1Condition"/>
        <link name="Activity2Condition"/>
    <links>

    <sequence name="Multi-Choice">
        <sources>
            <source linkName="Activity1Condition">
                <transitionCondition>"BooleanExpression1"
                </transitionCondition>
            </source>
            <source linkName="Activity2Condition">
                <transitionCondition>"BooleanExpression2"
                </transitionCondition>
            </source>
        </sources>
    </sequence>

    <sequence name="FirstBranch">
        <targets>
            <target linkName="Activity1Condition"/>
        </targets>
        <!-- Custom Activities -->
    </sequence>

    <sequence name="SecondBranch">
        <targets>
            <target linkName="Activity2Condition"/>
        </targets>
        <!-- Custom Activities -->
    </sequence>
</flow>

<!--Three-construct BPEL Solution-->
<flow>
    <if>
        <condition>"BooleanExpression1"</condition>
        <!--First Branch-->
    </if>
    <if>
        <condition>"BooleanExpression2"</condition>
        <!--Second Branch-->
    </if>
</flow>
```

WCP-7 Structured Synchronizing Merge: The convergence of multiple branches into a single branch which is enabled each time all of the active incoming branches have completed. This is an extension of the Simple Merge pattern to provide synchronization for a preceding Multi-Choice construct ([38], pp. 17–19).
Solutions: All the solutions presented for the Multi-Choice are structured and provide synchronization once all of the incoming branches have completed. Therefore, the support measures for the Structured Synchronizing Merge are identical to the ones for the Multi-Choice.

WCP-8 Multi-Merge: The convergence of multiple branches into a single branch, which is enabled each time an incoming branch completes ([38], pp. 19–20). This is a variation of the Structured Synchronizing Merge pattern where each single completion of an incoming branch triggers the execution of the subsequent branch.
WF: Due to its block-structuredness, a merging construct for multiple active branches in WF will only trigger subsequent activities exactly once for each completion of all incoming branches. There is no way to pass on the flow of control while branches preceding to the merging construct are still executing. In the graph-oriented flowchart modeling style, the Multi-Choice pattern cannot be implemented and consequently there is also no support for the Multi-Merge in this style.
WS-BPEL: Also for WS-BPEL, it is not possible for an activity to be executed multiple times through the convergence of multiple branches. A natural approach for implementing the pattern would be using `links`. However, an activity can only be executed after the status of all its incoming `links` has been evaluated, as defined by link semantics ([31], pp. 105–112). Thus, it can be executed only once, even if its incoming `links` become ready for evaluation at different times and multiple of them evaluate to true.

WCP-37 Acyclic Synchronizing Merge: The convergence of multiple control-flow branches into a single branch. Subsequent activities are enabled when all active incoming branches have completed. This is a special case of the Structured Synchronizing Merge where there is no single preceding Multi-Choice construct, but the divergence might have happened at different points in the process model. Also, the branches could be a subset of the branches from a preceding Multi-Choice. This pattern can be useful in unstructured process models. Cycles of incoming branches of the merging construct that would lead to multiple activations are not allowed ([38], pp. 69–71).
WF: A necessary prerequisite for this pattern is the ability to describe unstructured models where multiple branches can be activated. This is not possible in WF. The solution for the Multi-Choice described before is structured. There is no possibility to merge a subset of the different branches. Again, the flowchart modeling style does not allow for the activation of multiple branches. Consequently, this pattern cannot be implemented in WF.
WS-BPEL: WS-BPEL supports this pattern, but it must rely on the solution of the Multi-Choice based on `links` in the `flow` activity. The solution without `links` is structured and has the same limitations as the one in WF. Based on `links`, it is possible to merge multiple branches by having them target the same activity with outgoing `links`. This activity can only be enabled if all incoming `links` can be evaluated. This may be a subset of the branches in a `flow`. This provides direct support to the Acyclic Synchronizing Merge pattern. As for Sun BPEL, `links` are not supported, so this solution is not possible. Similar to WF, no other way to implement this pattern could be found and consequently Sun BPEL does not support

it. A solution where a subset of multiple branches can be merged requires the existence of at least three branches, two of which are merged. Merging a single branch is pointless. The edit distance of the solution depicted below is eleven.

Listing 11: Acyclic Synchronizing Patterns

```
1 <flow>
2     <links>
3         <link name="Activity1Condition"/>
4         <link name="Activity2Condition"/>
5         <link name="Activity3Condition"/>
6         <link name="Activity2Synchronization"/>
7         <link name="Activity3Synchronization"/>
8     <links>
9 
10     <sequence name="Multi-Choice">
11         <sources>
12             <source linkName="Activity1Condition">
13                 <transitionCondition>"BooleanExpression1"
14                 </transitionCondition>
15             </source>
16             <source linkName="Activity2Condition">
17                 <transitionCondition>"BooleanExpression2"
18                 </transitionCondition>
19             </source>
20             <source linkName="Activity3Condition">
21                 <transitionCondition>"BooleanExpression2"
22                 </transitionCondition>
23             </source>
24         </sources>
25     </sequence>
26 
27     <sequence name="FirstBranch">
28         <targets>
29             <target linkName="Activity1Condition"/>
30         </targets>
31         <!--Custom Activities-->
32     </sequence>
33     <sequence name="SecondBranch">
34         <targets>
35             <target linkName="Activity2Condition"/>
36         </targets>
37         <sources>
38             <source linkName="Activity2Synchronization"/>
39         </sources>
40         <!--Custom Activities-->
41     </sequence>
42     <sequence name="ThirdBranch">
43         <targets>
44             <target linkName="Activity3Condition"/>
45         </targets>
46         <sources>
47             <source linkName="Activity3Synchronization"/>
48         </sources>
49         <!--Custom Activities-->
50     </sequence>
51 
52     <sequence name="AcyclicSynchronizingMerge">
53         <targets>
54             <target linkName="Activity2Synchronization"/>
55             <target linkName="Activity3Synchronization"/>
56         </targets>
57         <!--Custom Activities-->
```

```
58      </sequence>
59  </flow>
```

WCP-38 General Synchronizing Merge: The convergence of multiple branches into a single branch. Subsequent activities are enabled when all active incoming branches have completed and it is not possible that an incoming branch will become active at any future time. This is a modification of the Acyclic Synchronizing Merge where there may be cycles in the process model that lead to multiple triggerings of the same incoming branch to the merging construct ([38], pp. 71/72).
Solutions: Neither in WF nor in WS-BPEL, the cycles that are necessary for a core aspect of this pattern can be implemented. The structuredness of these languages forbids the construction of such process models. `Links` in WS-BPEL are not allowed to create cycles ([31], p. 105).

WCP-9, WCP-28, WCP-29 Discriminator patterns: These patterns merge multiple distinct incoming control-flow branches into one subsequent branch. Their context is similar to the Simple Merge pattern. The subsequent branch is enabled (the Discriminator fires) as soon as the *first* one of the incoming branches, instead of *all* incoming branches, has completed. Completions of other incoming branches that take place after the Discriminator has fired do not result in the enabling of the subsequent branch, until all incoming branches have completed and the Discriminator is reset. There are three variants of this pattern. The *Structured Discriminator* ([38], pp. 20–23) lets all branches complete their execution even if the first one of them has completed. It is also allowed to cancel all incoming branches when the first one completes, although this violates one context criterion and thereby limits the traditional support measure to a rating of partial support. The *Blocking Discriminator* ([38], pp. 53–54) assumes a multi-threaded execution model, which is not safe. Like in a general purpose programming language, where it is possible that multiple threads execute identical lines of code and operate on identical objects and variables, it would be necessary to have multiple threads executing identical activity instances. This could lead to multiple concurrent completions of the same incoming branch to the Blocking Discriminator. Such multiple completions by different threads are blocked by the discriminator and retained until the discriminator has been reset. Finally, the *Cancelling Discriminator* ([38], p. 55) cancels all active branches, once the first of them completes.
WF: The Structured Discriminator is partially supported. Partial support is granted for this pattern if the branches are canceled once the first of them completes. This violates one context criterion which is why there is partial support. Canceling is possible using a `Parallel` activity and specifying a `CompletionCondition` which is set to evaluate to true if any of the branches completes. This can be achieved by letting all of the branches write to a shared variable indicating that they have completed. This is also a solution for the Cancelling Discriminator that provides direct support for this pattern. The Blocking Discriminator, on the other hand, is not supported. To implement the Blocking Discriminator, it would be necessary to integrate the Structured Discriminator in a multi-threaded environment. This is not possible using BAL activities and would only be achievable using `AsyncCodeActivities`. Hence, there is no support in WF for this pattern. The edit distance for the canceling discriminator is nine.
WS-BPEL: A straight-forward approach for implementing the Discriminator patterns would be using the `forEach` activity. Just as the `flow` activity, this loop executes its children in

parallel by setting its `parallel` attribute to `yes`. In contrast to the `flow` it also provides a `completionCondition` for specifying a premature end of the loop. If this condition evaluates to true, all remaining branches are canceled. This in principle resembles the structure of the solution for WF. However, there is no support for distinct branches. Each iteration of the loop will execute the same branch. Distinct branches are a core aspect of the pattern. A direct solution based on the `flow` activity is also not possible, as this activity lacks the necessary completion condition to provide a premature end of the execution. However, a more complex solution is possible. This solution is primarily aimed at the Cancelling Discriminator, but is valid for the Structured Discriminator as well. It is based on the mechanism for cancellation as discussed for several other patterns later on. First, a `flow` activity is necessary to provide multiple distinct concurrent branches. This `flow` needs to be embedded in a `scope` with an associated `faultHandler`. If a branch completes, instead of writing to a variable as in WF, the branch needs to throw a fault with the `throw` activity. This cancels the execution of all other branches and triggers the execution of the associated `faultHandler`. After the execution of this handler, subsequent activities can be executed normally. While the solution is valid for the Structured and Cancelling Discriminator, it requires too many constructs to provide support concerning the trivalent measure. The edit distance however is ten. As for the Blocking Discriminator, basically the same discussion as for WF applies. It is possible in WS-BPEL to have multiple threads execute the same branch definition in a parallel `forEach` activity. Still, each thread will execute a different branch instance represented by a different `scope` instance. Therefore the context of the Blocking Discriminator pattern cannot be implemented.

Listing 12: Discriminator Patterns

```
1 <!--WF Solution-->
2 <Parallel CompletionCodition="[ActivitiesExecuted \&gt; 0">
3     <Sequence>
4         <!-- Custom Activities -->
5         <Assign>
6             <!--Increment ActivitiesExecuted-->
7         </Assign>
8     </Sequence>
9
10    <Sequence>
11        <!-- Custom Activities -->
12        <Assign>
13            <!--Increment ActivitiesExecuted-->
14        </Assign>
15    </Sequence>
16 </Parallel>
17
18 <!--BPEL Solution-->
19 <scope>
20     <faultHandlers>
21         <catchAll>
22             <!--Discriminator fired-->
23         </catchAll>
24     </faultHandlers>
25     <flow>
26         <sequence>
27             <!-- Custom Activities -->
28             <throw />
29         </sequence>
30         <sequence>
31             <!-- Custom Activities -->
32             <throw />
```

```
33          </sequence>
34      </flow>
35  </scope>
```

WCP-30 Structured Partial Join, WCP-31 Blocking Partial Join, WCP-32 Cancelling Partial Join, WCP-33 Generalized AND-Join: The Partial Join patterns are extensions of the respective Discriminator patterns. A Discriminator passes the thread of control to subsequent activities when exactly *one* incoming branch has completed. A Partial Join passes the thread of control to subsequent activities when *N of M* incoming branches have completed. Again, the Structured Partial Join let's remaining incoming branches complete, the Cancelling Partial Join cancels all remaining branches and the Blocking Partial Join supports the Structured Partial Join in a multi-threaded environment by blocking further enablings of the partial join until it has been reset. The Generalized AND-Join works similar to the Blocking Partial Join. Like the Blocking Partial Join, it allows for multiple concurrent executions of the same branch. Instead of blocking multiple completions, it stores these completions and this way allows for multiple concurrent firings of the Partial Join ([38], pp. 58–65).
WF: The solution to the Discriminator patterns is the basis for the solution to these patterns. In fact, they are more or less identical. The only modification is that instead of having one branch complete to fire the Partial Join, the `CompletionCondition` must require more than one branch to complete to fire the construct. This can still be determined using only a single variable, because there is no concurrent write access to this variable due to the threading model of WF. It is assumed that a one-of-two partial join does not count as a valid solution, because this essentially is a Discriminator. Here a minimal solution is a two-of-three Partial Join (consequently only one of the three branches adds to the trivalent measure). Apart from this restriction, the same conditions apply and WF provides direct support for the Cancelling Partial Join and partial support for the Structured Partial Join. Due to the lack of built-in multi-threading, no support for the Blocking Partial Join and the Generalized AND-Join is provided. The edit distance of the solutions to the Structured and the Cancelling Partial Join is twelve. The solution is basically already outlined in listing 12 on the previous page.
WS-BPEL: Also WS-BPEL supports the Structured Partial Join and the Cancelling Partial Join based on the solutions to the Discriminator patterns. Here, more activities and variables are necessary. Each branch maintains a variable of type `int` that is initially zero. On completion, the branch must increment this variable for indicating that it has completed. Before throwing the fault, a branch must now test whether the critical amount of branches has been reached. This can be done using an `if` activity and summing up the values of the variables of all branches. If the number has reached the critical amount, a fault is thrown, thereby canceling the execution of the other branches. This solution works in the truly concurrent environment provided by WS-BPEL, because there are no concurrent write accesses to the same variables. Concurrent reads are unproblematic here. Again, this solution does not provide support according to the trivalent measure. The edit distance is 31. For the Blocking Partial Join and the Generalized AND-Join, the same restrictions as for WF apply. The WS-BPEL solution can be found in listing 13.

Listing 13: Partial Join Patterns in WS-BPEL

```
1 <scope>
2     <variables>
3         <variable name="Branch1Completed" type="int"/>
4         <variable name="Branch2Completed" type="int"/>
```

```
5          <variable name="Branch3Completed" type="int"/>
6      </variables>
7      <faultHandlers>
8          <catchAll>
9              <!--Partial Join fired-->
10         </catchAll>
11     </faultHandlers>
12     <sequence>
13         <assign>
14             <!-- Initialize variables with zero-->
15         </assign>
16
17         <flow>
18             <!--repeat this structure for every branch-->
19             <sequence>
20                 <!-- Custom Activities -->
21                 <assign>
22                     <!-- increment branch variable -->
23                 </assign>
24                 <if>
25                     <condition>"EnoughBranchesCompleted"</condition>
26                     <throw />
27                 </if>
28             </sequence>
29         </flow>
30
31     </sequence>
32 </scope>
```

WCP-41 Thread Merge, WCP-42 Thread Split: The patterns assume a threading model which is not safe. It is possible to spawn off a given number of threads that execute an identical control-flow branch. These threads can be merged into the main thread later on ([38], pp. 74–76). The difference of these patterns to other patterns related to concurrency, like the Parallel Split and the Synchronization patterns, is that the flow of control is not split into several concurrent branches. Instead, there is a single branch on which several threads execute. Such a spawn-off facility is necessary for other patterns related to multi-threading, like the Blocking Discriminator, Blocking Partial Join and Generalized AND-Join.
Solutions: As discussed in the context of the other multi-threading patterns, neither in WS-BPEL, nor in WF, it is possible to have multiple threads execute a single branch for the same process instance. Multiple threads for identical branches will always relate to multiple process instances. Consequently, neither of the languages supports these patterns.

Multiple Instances and Cancellation Patterns:

WCP-12 Multiple Instances without Synchronization, WCP-13 Multiple Instances with a priori Design-Time Knowledge, WCP-14 Multiple Instances with a priori Run-Time Knowledge: The Multiple Instances patterns concern multiple instances of an activity which are created simultaneously and run concurrently and independently of each other. The creation of the activity instances may also happen in sequential order. Once created, however, these activities must be able to execute concurrently. For the Multiple Instances without Synchronization, the activity instances do not need to be synchronized upon completion. Still, synchronizing these instances does not harm the validity of a solution. For the Multiple Instances with a priori Design-Time Knowledge, the number of activity instances is known at design time and they need to be synchronized upon completion. For the Multiple Instances

with a priori Run-Time Knowledge, this number is only known at run-time but before the execution of the multiple instances starts ([38], pp. 26–31).

WF: There is a single solution which is valid for all three patterns. This solution is based on the `ParallelForEach<T>` activity. This activity creates multiple instances of its child activity. For this, the number of instances must be known at runtime and is determined through the size of an input collection. The `ParallelForEach<T>` schedules one activity instance for execution for each of the items in the input collection. These activities are to be executed in a fashion similar to the `Parallel` activity. The activity completes when all its children complete, thereby providing synchronization. The type of the items of the input collection must be set at the level of the `ParallelForEach<T>`. The activity to be executed multiple times is then placed in the `ActivityAction` child element of the `ParallelForEach<T>` and the type of this child element needs to be set to the same type as the `ParallelForEach<T>`. The edit distance of this solution is six and it grants direct support.

WS-BPEL: A solution with the `forEach` activity in WS-BPEL works similar to the solution in WF. The difference is mainly that, instead of a collection, the `forEach` activity requires the definition of the starting and ending value of a counter variable and it has to be set to run in parallel explicitly through the modification of an attribute. Child activities must run inside a `scope`. As multiple instances of this `scope` run in parallel, it has to be made sure that no race conditions occur when manipulating `variables`. The edit distance of this solution is seven and it provides direct support. The `scope` activity does not count to the trivalent measure, as it is an integral part of the `forEach` activity. Concurrent execution of the different activity instances is crucial for these patterns. Sun BPEL does not support the parallel attribute of the `forEach` activity and therefore cannot benefit from this solution. Also embedding multiple activities with equal definitions in a `flow` activity does not form a valid solution. Even if the activity definitions are syntactically equal, they are still distinct definitions. It is a core aspect of this pattern that multiple instances of an identical definition are executed. So, a solution with a `flow` activity conforms to the execution trace of the pattern, but it does still not fulfill its core aspects. Here, a more complex workaround solution is necessary. This solution involves two processes. One of the processes encapsulates the multiple instance activity. The other process contains an `invoke` activity embedded in a looping activity and invokes the first process multiple times asynchronously. Thus, the creation of the multiple activity instances happens sequentially, but once started, they execute concurrently. The process realizing the multiple instance activity may be identical to a process stub and therefore does not influence the support measures. The edit distance of this solution is twelve. As it requires a looping activity and an `invoke` activity, the solution only provides partial support. However, this solution does not provide synchronization and thereby only supports the Multiple Instances without Synchronization pattern. To achieve synchronization, also synchronous interaction is necessary. Synchronous interaction necessarily must be embedded in a concurrent environment; otherwise, the activity instances would be created and executed in a strictly sequential fashion which violates a core aspect of the Multiple Instances patterns. Concurrent execution in Sun BPEL can still be achieved by embedding multiple `invoke` activities in a `flow` activity. This is possible here, as the multiple instance activity is found in the invoked process and thus, still the identical activity definition is executed multiple times. As the definition in the flow can only be changed at design-time, this solution only implements the Multiple Instances with a priori Design-Time Knowledge pattern. It provides partial support and an edit distance of 19. Multiple Instances with a priori Run-Time Knowledge cannot be implemented in Sun BPEL.

Listing 14: Multiple Instances Patterns

```
1 <!--WF Solution-->
2 <ParallelForEach Values="[InputCollection]" TypeArguments="Int32">
3     <ActivityAction TypeArguments="Int32">
4         <!-- Multiple Instance Activity-->
5     </ActivityAction>
6 </ParallelForEach>
7
8 <!--WS-BPEL Solution-->
9 <forEach parallel="yes" counterName="ForEachCounter">
10     <starCounterValue>0</startCounterValue>
11     <finalCounterValue>3</finalCounterValue>
12     <scope>
13         <!-- Multiple Instance Activity-->
14     </scope
15 </scope>
16
17 <!--Sun BPEL Solution without Synchronization-->
18 <forEach counterName="ForEachCounter">
19     <starCounterValue>0</startCounterValue>
20     <finalCounterValue>3</finalCounterValue>
21     <scope>
22         <invoke operation="ExecuteActivityInstance" inputVariable="ActivityInput" />
23     </scope
24 </scope>
25
26 <!--Sun BPEL Solution with a priori Design-Time Knowledge-->
27 <flow>
28     <invoke operation="ExecuteActivityInstance" inputVariable="ActivityInput"
           outputVariable="ActivityOutput1"/>
29     <invoke operation="ExecuteActivityInstance" inputVariable="ActivityInput"
           outputVariable="ActivityOutput2"/>
30 </flow>
```

WCP-15 Multiple Instances without a priori Run-Time Knowledge: This Multiple Instances pattern poses the additional restriction that the number of instances required may not be known at any time before the execution of the multiple instances starts. New instances may be triggered even during the execution of other instances ([38], pp. 31–33). The number of required instances becomes known at some time during the execution of the multiple instance activity. When all required activity instances complete, the multiple instances activity completes and thereby synchronizes the different instances.
Solutions: No solution to this pattern in either of the languages could be found. In the case of WS-BPEL, the problem is that the `forEach` activity does not allow to modify the counter variable. Precisely speaking, it is possible to modify the variable, but these modifications are overwritten in each iteration. Thus, it is not possible to modify the amount of instances during the execution of the multiple instance activity. As for WF, the `ParallelForEach<T>` activity reads the input collection when starting and stores its state internally. During its execution, the input collection can be modified, but this does not affect the `ParallelForEach<T>`. That means that it is not possible in WS-BPEL or WF to modify the number of required instances during their execution. A solution that uses messaging activities and creates the multiple instances sequentially similar to the solution for the Multiple Instances without Synchronization pattern in Sun BPEL is not possible. This solution would in principle allow to modify the number of instances required during their execution, but it does not allow to synchronize these instances. As discussed, this solution is only able to provide synchronization by sequentializing the execution of the activity instances. The aspect of concurrent execution however is core to

all Multiple Instances patterns and therefore this solution cannot be used here. For a solution with synchronization, a parallel creation of the multiple instances is necessary and this is only provided by the two activities discussed above or by fixing the number of activity instances at design-time.

WCP-20 Cancel Case: A complete process instance is canceled. All executing activity instances are stopped and the process instance is recorded as having completed unsuccessfully ([38], pp. 39–41). As cancelation is not reasonable as the best-case behaviour of a realistic process, it is supposed to take place in a conditional branch in the executable processes. This is achieved by embedding the cancellation mechanism in an Exclusive Choice construct.
WF: There is a direct solution to this pattern in WF. The desired effect is achieved with the `Terminate` activity. This activity raises an exception, terminates all activities that are currently executing and transitions the process instance to the `Faulted` state. The `terminate` activity requires the specification of an exception to be thrown and needs to be embedded in an `If` activity for implementing the Exclusive Choice. The edit distance of the solution is four.
WS-BPEL: In WS-BPEL, similar results can be achieved with the `exit` and the `if` activity. The `exit` activity also leads to the immediate termination of the current process instance without any termination, fault or compensation handling. As the specification of a fault is not required, the edit distance of the solution is three.

Listing 15: Cancel Case Pattern

```
1 <!--WF Solution-->
2 <If Condition="BooleanExpression">
3     <If.Then>
4     <TerminateWorkflow Exception="New Exception()"/>
5     </If.Then>
6 </If>
7 <!--BPEL Solution-->
8 <if>
9     <condition>"BooleanExpression"</condition>
10     <exit/>
11 </if>
```

WCP-19 Cancel Activity, WCP-25 Cancel Region, WCP-26 Cancel Multiple Instance Activity: Activity instances in a process can be canceled. This can either happen before their execution starts or while they are executing. In the first case, they will never be executed. In the second case, their execution is stopped immediately. For the Cancel Activity pattern, it is sufficient if a single activity can be canceled ([38], pp. 37–39). For the Cancel Region pattern, any arbitrary set of activities can be canceled. The activities to be canceled need not be directly connected ([38], pp. 49/50). For the Cancel Multiple Instance Activity pattern, the activity to be canceled is a solution for one of the Multiple Instances patterns. The number of instances to be created may be known at design-time or not need to be synchronized. If the multiple instance activity is canceled during its execution, the activity instances that have not completed yet are withdrawn. The others remain unaffected ([38], pp. 50/51). All patterns have similar solutions in WS-BPEL and WF. Only in Sun BPEL, a more complex solution to WCP-26 is necessary.
WF: WF provides a `CancellationScope` activity that may contain another activity. If this activity is canceled, the associated `CancellationHandler` is invoked. This resembles the pattern. The problem is that there is no built-in activity that is able to trigger the cancellation. The cancellation of an activity must be triggered programmatically or by the operating envi-

ronment and this is out of scope for the analysis at hand. Nevertheless, a workaround that achieves identical results can be used. Instead of a `CancellationScope`, a `TryCatch` activity can be used. This activity contains the activity that can be canceled in its `Try` element. At any point during the execution of the `Try`, the cancellation can be performed by throwing an exception using the `Throw` activity (embedded in the conditional branch of an `If` activity). This immediately aborts the execution of the `Try`. The exception needs to be caught in a `Catch` and the control-flow may proceed normally after the `TryCatch`. This resembles standard exception handling as in Java or C#. It works independently of whether there is a single activity to be canceled, a set of activities, or a Multiple Instances activity. If a set of activities is to be canceled, the set of activities is not arbitrary, but part of a connected graph. This violates one context criterion for the Cancel Region pattern, limiting the trivalent support measure for this pattern to a maximum of partial support. For the Cancel Multiple Instance Activity pattern, the activity embedded in the `Try` must be a solution for one of the Multiple Instances patterns. In case one of the activity instances throws an exception, all of the other instances that did not complete yet can be canceled. Completed activities remain completed, but the overall multiple instance activity transits to the `Faulted` state. As for the amount of constructs, essential to this pattern is the combination of `TryCatch` activity and `Throw` activities. So the solution provides partial support. The edit distance is nine for the Cancel Activity and Cancel Region pattern. As the Cancel Multiple Instance Activity pattern must contain a multiple instance activity, the edit distance is slightly higher with a value of 14.

WS-BPEL: WS-BPEL supports the patterns in a similar fashion using the `throw` and `if` activities, as well as `faultHandlers` attached to a `scope` activity. An activity can be canceled by throwing a fault before or during its execution. The activity, as well as the remaining contents of the enclosing `scope`, are then skipped and the logic in the corresponding `catch` or `catchAll` element of the `faultHandler` attached to the `scope` is executed. Like for WF, this solution provides partial support for the Cancel Activity and the Cancel Region pattern. The edit distance for these patterns amounts to eight. Again, the Cancel Multiple Instance Activity pattern is more expensive and has different values for WS-BPEL and Sun BPEL, because Sun BPEL supports the Multiple Instances without Synchronization pattern with a more complex solution. For WS-BPEL, the edit distance to this pattern amounts to 13. For Sun BPEL, the solution needs to be extended with several constructs. Here, any solution to the Multiple Instances patterns requires several process instances. One process instance for initiating the multiple activity instances and another process instance for each of the activity instances (cf. page 59). Therefore, canceling activity instances corresponds to the cancellation of the process instances that run these activity instances. This cancellation needs to be triggered from the initiating process instance. The trigger is accomplished by sending a message to the respective process instances. This message transmission is needed in the case of cancellation, so it needs to be positioned in the `faultHandler`. The process instances that execute activity instances need to be ready to receive such a cancellation message at any time. This can be accomplished by attaching an `eventHandler` to their process `scope`. This handler must immediately terminate the process instance which is achieved by an `exit` activity. Obviously, this solution is quite complex and requires stateful conversations between multiple process instances. Too many constructs are needed to provide support according to the trivalent measure. The solution has an edit distance of 55.

Listing 16: Cancellation Patterns in WF and WS-BPEL

```
<!--WF Solution-->
<TryCatch>
    <TryCatch.Try>
        <Sequence>
            <!-- Custom activities-->
            <If Condition="BooleanExpression">
                <If.Then>
                    <Throw Exception="[New Exception()]"/>
                </If.Then>
            </If>
            <!--Custom activities that may be canceled-->
        </Sequence>
    </TryCatch.Try>
    <TryCatch.Catches>
        <Catch TypeArguments="Exception">
            <ActivityAction TypeArguments="Exception" />
        </Catch>
    </TryCatch.Catches>
</TryCatch>

<!--BPEL Solution-->
<scope>
    <faultHandlers>
        <catchAll>
            <!--Cancellation activity-->
        </catchAll>
    </faultHandlers>
    <sequence>
        <!--Custom activities-->
        <if>
            <condition>"BooleanExpression"</condition>
            <throw faultName="fault"/>
        </if>
        <!--Custom activities that may be canceled-->
    </sequence>
</scope>
```

Listing 17: Cancel Multiple Instance Activity Pattern in Sun BPEL

```
<!--Multiple Instances main process-->
<scope>
    <faultHandlers>
        <catchAll>
            <sequence>
                <!--repeat for every activity instance-->
                <invoke operation="CancelActivityInstance"/>
        </catchAll>
    </faultHandlers>
    <flow>
        <!--repeat for every activity instance required-->
        <invoke operation="ExecuteActivityInstance" />
        <!--When cancellation is necessary-->
        <if>
            <condition>"BooleanExpression"</condition>
            <throw faultName="fault"/>
        </if>
    </flow>
</scope>

<!--Process for single activity instance-->
<eventHandlers>
    <onEvent operation="CancelActivityInstance">
```

```
24          <scope>
25              <exit />
26          </scope>
27      </onEvent>
28  </evenHandlers>
29  <sequence>
30      <receive operation="ExecuteActivityInstance" />
31      <!--Multiple Instance activity-->
32  </sequence>
```

WCP-27 Complete Multiple Instance Activity: It is possible to create multiple instances of an activity the number of which may be known at design-time. The activity instances need to be synchronized on completion before the control-flow can proceed to subsequent activities. It is possible to forcibly complete the multiple instance activity. This could be triggered by one of the activity instances or from another point of the process. Remaining activity instances are canceled, but the multiple instance activity is recorded as having completed successfully ([38], pp. 51–53). This pattern resembles the Cancel Multiple Instance Activity pattern. The main difference is that for this pattern the Multiple Instances activity is recorded as having completed successfully, while for the cancellation pattern, it is recorded as having completed unsuccessfully.

WF: For this pattern, a rather elegant solution is available. The `ParallelForEach<T>` activity provides a `CompletionCondition`. As soon as this condition evaluates to true, all activities that have not completed yet are canceled. In this case, the `ParallelForEach<T>` activity transitions to the `Completed` state, as opposed to the `Faulted` state which was the case for the Cancel Multiple Instance Activity pattern. The solution provides direct support with an edit distance of ten.

WS-BPEL: In WS-BPEL, a very similar solution is possible. Here, the `forEach` activity also provides a `completionCondition`. This condition allows for specifying the number of activity instances that have to complete for the Multiple Instances activity to complete. The solution for WS-BPEL provides direct support with an edit distance of eight. This solution is not realizable in Sun BPEL, as it does not support a parallel `forEach` activity. There, the Multiple Instances with a priori Design-Time Knowledge pattern builds on inter-process communication (cf. page 59). Of course, it is possible to cancel the process instances that are executing activity instances and the `flow` activity that initiated them. Such a solution also conforms to the execution trace of the pattern. The point is that any solution that employs such manual cancellation would leave the `flow` activity in a canceled state, as opposed to a completed state. The difference between such a canceled or completed state is not visible when executing a process instance, but it is relevant to this pattern. A solution that does not provide some mechanism that leads to a graceful completion of the `flow` would rather be a solution to the Cancel Multiple Instance Activity pattern. Sun BPEL does not provide a mechanism for the graceful completion of the `flow` activity in spite of its cancellation and does therefore not support this pattern.

Listing 18: Complete Multiple Instance Activity Pattern

```
1 <!--WF Solution-->
2 <ParallelForEach TypeArguments="Int32" CompletionCondition="[ShouldCompleteActivity]"
      Values="InputCollection">
3     <ActivityAction TypeArguments="Int32">
4         <Sequence>
5             <!-- Multiple instance activity -->
```

```
6             <Assign>
7                 <!-- Set ShouldCompleteActivity to true -->
8             </Assign>
9         </Sequence>
10    </ActivityAction>
11 </ParallelForEach>
12
13 <!--WS-BPEL Solution-->
14 <forEach parallel="yes" counterName="ForEachCounter">
15     <startCounterValue>0</startCounterValue>
16     <finalCounterValue>3</finalCounterValue>
17     <completionCondition>
18         <branches>2</branches>
19     </completionCondition>
20     <scope>
21             <!-- Multiple instance activity -->
22     </scope>
23 </forEach>
```

WCP-34 Static Partial Join for Multiple Instances, WCP-35 Cancelling Partial Join for Multiple Instances, WCP-36 Dynamic Partial Join for Multiple Instances: These patterns form a combination of the Partial Join and the Multiple Instances patterns where the multiple instance activity may complete after a given number of activity instances have completed. In other words, the patterns are a special case of the Complete Multiple Instance Activity pattern, where the completion of the multiple instance activity may only be triggered through completed activity instances. For the first two patterns, the number of instances required is known at a priori run-time and the instances need to be synchronized. For the Static Partial Join for Multiple Instances, the multiple instance activity may simply complete after a number of instances have completed. Activity instances that are still executing may finish their execution normally. For the Cancelling Partial Join for Multiple Instances, the remaining instances are canceled. The Dynamic Partial Join for Multiple Instances builds on the Multiple Instances without A Priori Run-Time Knowledge pattern. The number of instances required is not known until the completion of the multiple instance activity ([38], pp. 65–69).
WF: The solution to the Complete Multiple Instance Activity pattern depicted in listing 18 on the previous page also provides a solution to the Static and the Cancelling Partial Join for Multiple Instances patterns. Here, already, a multiple instance activity completes after one activity instance has completed. The solution requires a slight modification. The `CompletionCondition` must be based on a counter instead of a boolean variable and each activity instance needs to increment this counter. In case a predefined number of instances completes, the whole multiple instance activity should complete. Similar to the Discriminator and the Partial Join patterns, this solution provides direct support for the Cancelling Partial Join for Multiple Instances and partial support for the Static Partial Join for Multiple Instances. The edit distance is equal to the edit distance for the solution to the Complete Multiple Instance Activity pattern. The Dynamic Partial Join for Multiple Instances is not supported as Multiple Instances without A Priori Run-Time Knowledge cannot be implemented in WF.
WS-BPEL: WS-BPEL is able to provide support for some of the patterns with more elegant solutions than for the Discriminator and Partial join patterns. This is the case because here, identical branches are required instead of distinct branches. Therefore, the `forEach` activity, which also allows for a `completionCondition`, may be used. This does not apply to Sun BPEL which does not implement a parallel `forEach`. Furthermore, as all of the patterns require Mul-

tiple Instances with a priori Run-Time Knowledge, Sun BPEL is not able to support any of them. Just as for WF, the WS-BPEL solution is identical to the solution for the Complete Multiple Instance Activity pattern. The Static Partial Join for Multiple Instances pattern is partially supported, the Cancelling Partial Join for Multiple Instances pattern is directly supported, and the Dynamic Partial Join for Multiple Instances pattern is not supported, as there is no support for the Multiple Instances without a priori Run-Time Knowledge pattern. The edit distances again amount to eight.

State-based Patterns:

WCP-16 Deferred Choice: This pattern describes the divergence of the control-flow into two or more branches exactly one of which is enabled. The choice of the branch is not made by a logical expression, like for the Exclusive Choice pattern, but through interaction with the operating environment, such as an event or a trigger ([38], pp. 33/34). For a choice, at least two alternatives are required in a valid solution.
WF: This pattern can be implemented using the `Pick` activity. The `PickBranch` activities contained in the `Pick` comprise the triggers which can for example be inbound messaging activities such as `Receive` activities, that serve as event-based choice. The `PickBranch` whose trigger completes first is selected. This solution has an edit distance of nine. As a combination of activities, `Pick` and `Receive`, is necessary, the solution does only provide partial support.
WS-BPEL: The pattern is directly supported by the `pick` activity containing multiple `onMessage` activities. Each `onMessage` activity represents one branch of execution and which of the branches is executed depends on the message which is received first. The edit distance of this solution is eight. Because the `onMessage` counts as an integral part of the `pick` activity, it does not influence the trivalent support measure which rates as direct support. This solution is slightly more costly in Sun BPEL. In WSDL 1.1, it is not possible to define `operations` that do have neither input, nor output messages. However, operations without input and output can be implemented by using a `messageType` as input that does not define any content. If this is the case, it is possible to ignore such an input in WS-BPEL. As uncovered during implementing the solution, Sun BPEL is not able to deal with empty messages. Here, the reception of an empty message causes an exception in the engine that leads to the termination of the process instance. Consequently, in Sun BPEL, input messages with content have to be used and the `inputVariable` of the `onMessages` have to be configured. This increases the edit distance for Sun BPEL to ten.

Listing 19: Deferred Choice Pattern

```
1 <!--WF Solution-->
2 <Pick>
3     <PickBranch>
4         <PickBranch.Trigger>
5             <Receive OperationName="Choice1" CanCreateInstance="True"/>
6         </PickBranch.Trigger>
7         <!-- Choice1 Activity -->
8     </PickBranch>
9     <PickBranch>
10        <PickBranch.Trigger>
11            <Receive OperationName="Choice2" CanCreateInstance="True"/>
12        </PickBranch.Trigger>
13        <!-- Choice2 Activity -->
14    </PickBranch>
15 </Pick>
```

```
16
17 <!--BPEL Solution-->
18 <pick createInstance="yes">
19     <onMessage operation="Choice1"/>
20         <!-- Choice1 Activity -->
21     </onMessage>
22     <onMessage operation="Choice2"/>
23         <!-- Choice2 Activity -->
24     </onMessage>
25 </pick>
```

WCP-17 Interleaved Parallel Routing: This pattern describes the definition of a partial ordering defined on a set of activities. The activities are executed in arbitrary order, except for the restrictions in the partial ordering. The activities may not execute concurrently. The possibility to define the partial ordering on activities is a core aspect of this pattern. A solution that omits this partial ordering is instead a solution to the Interleaved Routing pattern. Moreover, a solution that enumerates all possible execution sequences (and therefore adheres to the execution trace of the pattern) and then chooses one of these sequences via a Deferred Choice construct does not reflect the dynamic nature of this pattern and thus violates a core aspect ([38], pp. 34–36).
WF: Support for different interleavings of the execution of several activities is provided through the `Parallel` activity. However, there is no possibility to define a partial ordering between arbitrary activities of different branches inside the parallel activity. A solution that lists all possible execution sequences would be possible, but is not considered to be valid for this pattern, due to its static nature.
WS-BPEL: A way of implementing this pattern in WS-BPEL is by using isolated `scopes`. These are `scope` activities with their `isolated` attribute set to true. Isolated `scopes` have exclusive read and write access to shared resources such as `variables` or `partnerLinks` and thus may not be executed concurrently. So, the activities to be interleaved can be embedded in different isolated scopes in a `flow` activity and need to access shared resources before and after an interleaved activity is executed. This implements the behaviour of a semaphore that ensures that no other activity is executed at the same time. The partial ordering can be defined using `links`. A minimal solution built in this manner has an edit distance of 16. The pattern requires the existence of at least two activities, because otherwise there would be no interleaving of different activities. The solution presented here consists of a `flow` and `scope` activities, so it provides partial support. Sun BPEL does not support isolated scopes, and so this solution can also not be used. Like for WF, no other solution could be found.

Listing 20: Interleaved Parallel Routing Pattern in WS-BPEL

```
1 <flow>
2     <links>
3         <link name="Ordering"/>
4     </links>
5
6     <scope name="Scope1" isolated="yes">
7         <sources>
8             <source linkName="Ordering"/>
9         </sources>
10        <sequence>
11            <assign>
12                <!-- Entry: Access shared variable -->
13            </assign>
14            <!-- Interleaved Activity -->
```

```
15            <assign>
16                <!-- Exit: Access shared variable -->
17            </assign>
18        </sequence>
19    </scope>
20
21    <scope name="Scope2" isolated="yes">
22        <targets>
23            <target linkName="Ordering"/>
24        </targets>
25        <sequence>
26            <assign>
27                <!-- Entry: Access shared variable -->
28            </assign>
29            <!-- Interleaved Activity -->
30            <assign>
31                <!-- Exit: Access shared variable -->
32            </assign>
33        </sequence>
34    </scope>
35 </flow>
```

WCP-18 Milestone: This pattern describes a situation where an activity is only enabled, given the process instance is in a certain state, also called milestone. If it leaves this state, the activity is no longer executable ([38], pp. 36/37).
WF: There is no direct representation of state in WF, but workaround solutions are possible. One workaround for a special variant of a milestone, a deadline, is possible based on the solution to the Deferred Choice pattern. Just as for the Deferred Choice, there is a `Pick` activity that contains two `PickBranches`. The trigger of the first branch contains the deadline. This is a `Delay` activity where the `Duration` is set to expire when the deadline is reached. The trigger of the second `PickBranch` contains a `Receive` activity that expects the signal to execute the milestone activity. The body then comprises this activity. The edit distance of this solution is eleven. The `Pick` and the `Receive` activities count to the trivalent measure. Therefore, this solution grants partial support.
WS-BPEL: Basically, the same argumentation and a similar solution is possible in WS-BPEL. Here it is accomplished with an `onAlarm` and an `onMessage` activity. In spite of the fact that WS-BPEL requires the definition of body activities in the two branches, support measures are identical in both languages.

Listing 21: Milestone Pattern

```
1 <!--WF Solution-->
2 <Pick>
3     <PickBranch DisplayName="Deadline">
4         <PickBranch.Trigger>
5             <Delay Duration="DurationUntilDeadlineExpiration"/>
6         </PickBranch.Trigger>
7     </PickBranch>
8     <PickBranch DisplayName="Milestone">
9         <PickBranch.Trigger>
10            <Receive OperationName="ExecuteMilestoneActivity">
11        </PickBranch.Trigger>
12        <!-- Milestone activity -->
13    </PickBranch>
14 </Pick>
15
16 <!--BPEL Solution-->
```

```
17 <pick>
18     <onAlarm name="Deadline">
19         <for>"DeadlineDate"</for>
20     </onAlarm>
21     <onMessage operation="ExecuteMilestoneActivity">
22         <!-- Milestone activity -->
23     </onMessage>
24 </pick>
```

WCP-39 Critical Section: Two or more concurrent subgraphs of a process are identified as critical sections. These critical sections may not be executed at the same time and if one of the critical sections starts to execute, it must complete before the other one can begin ([38], pp. 72/73). A solution to this pattern requires the use of a minimum of two branches, otherwise there would be no competing critical sections. This pattern clearly addresses the availability of some synchronization primitive in a language.
WF: The pattern can be implemented using the `TransactionScope` activity and its `IsolationLevel` property. Each critical section is identified by a `TransactionScope` activity. At least two such scopes that operate on common data are necessary. To enable concurrent execution, these activities need to be embedded in a `Parallel` activity. The different critical sections must share some variable as mutual exclusion property. If they are granted the access to the shared variable, they will not release it until their completion and thus block any other critical section from executing. So, the `TransactionScopes` begin by trying to write to the shared variable and thus determine whether they are allowed to enter the critical section. By setting the `IsolationLevel` of the `TransactionScope` activities to `Serializable` (which is its default) no other transaction can write any data that is used by this transaction until the transaction is committed. If any transaction tries to start while another one is in progress, it is prevented from commencing until the other one is committed. Altogether, the solution provides partial support with an edit distance of nine.
WS-BPEL: The solution for this pattern is very similar to the solution in WF. The critical sections are encapsulated in isolated `scopes`. The isolated `scopes` access a shared variable to accomplish mutual exclusion. Here however, they must access the shared variable before and after the critical section. This is necessary to first optain the lock on the variable and then keep it until the end of the execution of the critical section. WS-BPEL automatically releases the lock if there are no more subsequent references to the shared variable in the same isolated `scope`. As for WF, this solution also provides partial support with an edit distance of 15. Sun BPEL does not support isolated `scopes`. Here, no built-in synchronization mechanism is available. Still, the mutual exclusion problem is one of the most basic problems of distributed systems and several algorithms are available for solving it. One of the first and well-known solutions to this problem is the Dekker algorithm [14][18]. This algorithm provides mutual exclusion for two processes using a shared memory space. With basic control-flow activities, such as `while` and `if`, and boolean and integer `variables`, the algorithm can also be implemented in Sun BPEL. The solution is obviously more complex than the previous ones and does not provide support according to the trivalent measure. Its edit distance is 40.

[18]The reader is assumed to be familiar with this algorithm. Therefore a discussion on its functioning is not provided here

Listing 22: Critical Section Pattern

```
<!--WF Solution-->
<Parallel>
    <!--Repeat this structure for every Critical Section-->
    <TransactionScope>
        <Sequence>
            <Assign>
                <!-- Access shared variable-->
            </Assign>
            <!-- Critical Section Activities -->
        </Sequence>
    </TransactionScope>
</Parallel>

<!--WS-BPEL Solution-->
<flow>
    <!--Repeat this structure for every Critical Section-->
    <scope isolated="yes">
        <sequence>
                        <assign>
                <!-- Entry: Access shared variable -->
            </assign>
            <!-- Critical Section Activities -->
            <assign>
                <!-- Exit: Access shared variable -->
            </assign>
        </sequence>
    </scope>
</flow>

<!--Sun BPEL Solution-->
 <variables>
        <variable name="turn" type="int"/>
        <variable name="flag2" type="boolean"/>
        <variable name="flag1" type="boolean"/>
</variables>
<flow>
    <!--Repeat this structure for every Critical Section with different flags-->
    <sequence>
        <!--pre-protocol-->
        <!--assign true to flag1-->
        <while>
            <condition>$flag2</condition>
            <if>
                <condition>$turn != 0</condition>
                <sequence>
                    <!--assign false to flag1-->
                    <while>
                        <condition>$turn != 0</condition>
                            <!--busy waiting-->
                    </while>
                    <!--assign true to flag1-->
                </sequence
            </if>
        </while>
        <!--Critical section activities-->
        <!--post-protocol-->
        <!--assign false to flag1 and alter turn-->
    </sequence>
</flow>
```

WCP-40 Interleaved Routing: A set of activities can be executed in any order. At any time, only a single activity may execute ([38], pp. 72/73). This is a modification of the Interleaved Parallel Routing pattern where there is no requirement for the definition of a partial ordering between the interleaved activities.
WF: This pattern can be implemented using the same mechanism as for the Critical Section pattern. Each of the activities to be interleaved can be encapsulated in a critical section. The difference to the Interleaved Parallel Routing pattern here is that no order needs to be defined. Thus WF is also able to cover all core aspects. This implies again partial support and an edit distance of nine.
WS-BPEL: Just as for WF, also in WS-BPEL the solution for the Critical Section pattern can be used. The definition of an ordering is not necessary, therefore also `links` (as for the Interleaved Parallel Routing pattern) need not be used, but the `scopes` may simply be embedded in the `flow` activity. The `scopes` must still be isolated and write to shared variables. The solution provides partial support and the edit distance is 15. Sun BPEL supports this pattern again using the Dekker algorithm [14]. This solution provides no direct support with an edit distance of 40.

Iteration Patterns:

WCP-21 Structured Loop: An activity can be executed repeatedly, based on a logical condition. There is a single entry and exit point to the execution cycle ([38], pp. 42/43).
Solutions: All of the looping activities in WS-BPEL or WF are structured and directly support this pattern. Apart from the looping structure itself, a solution requires a variable that is used to control the repeated execution and an activity that manipulates this variable. The WF looping activities are `ForEach<T>`, `ParallelForEach<T>`, `DoWhile` and `While`. The WS-BPEL looping structures are `while`, `forEach` and `repeatUntil`. The solutions here are implemented using looping activities of the `while` type. They provide direct support and the edit distances for both languages are equal to five.

Listing 23: Structured Loop Pattern

```
1 <!-- WF Solution -->
2 <While Condition="[LoopCounter &gt 0]">
3     <Sequence>
4         <!-- Custom activity -->
5         <Assign>
6             <!--Decrement LoopCounter-->
7         </Assign>
8     </Sequence>
9 </While>
10
11 <!-- BPEL Solution -->
12 <while>
13     <condition>$LoopCounter &gt; 0</condition>
14     <sequence>
15         <!-- Custom activity -->
16         <assign>
17             <!-- Decrement LoopCounter-->
18         </assign
19     </sequence>
20 </while>
```

WCP-10 Arbitrary Cycles: A process model contains a cycle which has more than one entry or exit point ([38], pp. 24/25). The implementer of this pattern must make sure that this structure does not contain life- or deadlocks. Block-structured languages are generally unable to directly support this pattern with a dedicated construct [21]. However, as demonstrated in [56], it is possible to untangle unstructured loops and map them to a structured representation. Figure 6 demonstrates an abstract model for an arbitrary cycle involving the activities B and C. This model is extracted from Figure 3 on page 72 of [56]. It is an excerpt of the use case

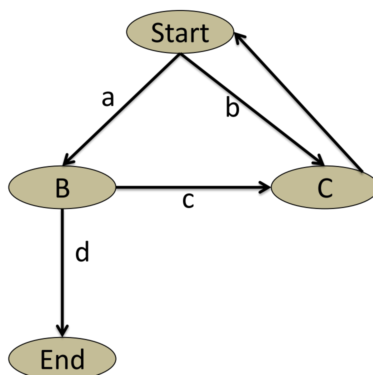


Figure 6: Model of an arbitrary cycle extracted from [56], Figure 3, page 72.

of this paper that is sufficient to represent a minimal arbitrary cycle. Small letters represent conditions for the activation of the associated control-flow edges. Based on the evaluation of these conditions, the activities B and C are executed repeatedly and in any order. The arbitrary cycle is formed by the repeated executin of C. Both, the start node and activity B can serve as entry point to this cyt'cle. The following solutions in WF and WS-BPEL implement this model.
WF: In WF, the flowchart style is ideally suited to implement the Arbitrary Cycles pattern and no untangling of loops is required. A cycle can be created using a `FlowDecision` as the head of a loop. This head points to the activities B and C. C unconditionally points back to the head of the loop. B is followed by another `FlowDecision` that determines whether C should be executed next or whether the cycle should be terminated. Please notice that Figure 4 on page 15 visualizes this structure. As this solution involves a `FlowChart` and several `FlowDecision` activities, it achieves a rating of no direct support for the trivalent measure. Its edit distance however is 17.
WS-BPEL: WS-BPEL is mainly block-structured, so there is no direct way to describe loops with more than one entry or exit point. An exception to this block-structured style are `links` in a `flow` activity. However, link semantics specify that `links` must not create cycles ([31], p. 105), thereby also prohibiting the implementation of Arbitrary Cycles. So, the pattern must be implemented by untangling the unstructured loop. This involves a `while`, two `if` activities, and a replication of activity C. The `while` activity controls the complete cycle between start and end. Based on the evaluation of conditions, either C alone can be executed, or B followed by C and the repetition of the overall cycle, or B followed by the end of the cycle. This exactly

reflects the behaviour defined by the abstract model and thereby implements the arbitrary cycle depicted in it. Like the solution for WF, this solution does not qualify for support according to the trivalent measure. Its edit distance amounts to 18.

Listing 24: Arbitrary Cycles Pattern

```
<!-- WF Solution -->
<Flowchart>
    <Flowchart.StartNode>
        <Reference>LoopHead</Reference>
    </Flowchart.StartNode>

    <FlowStep Name="StepToC">
        <!-- Activity C -->
        <FlowStep.Next>
            <FlowDecision Name="LoopHead" Condition="ShouldGoToC" True="{Reference StepToC}
                ">
                <FlowDecision.False>
                    <FlowStep Name="StepToB">
                        <!--Activity B-->
                        <FlowStep.Next>
                            <FlowDecision Name="DecisionAfterB" Condition="ShouldGoToC"
                                True="{Reference StepToC}">
                                <FlowDecision.False>
                                    <FlowStep Name="EndOfLoop" />
                                </FlowDecision.False>
                            </FlowDecision>
                        </FlowStep.Next>
                    </FlowStep>
                </FlowDecision.False>
            </FlowDecision>
        </FlowStep.Next>
    </FlowStep>

</Flowchart>

<!-- BPEL Solution -->
<while>
    <condition>$ShouldExecuteLoop</condition>
    <if>
        <condition>$ShouldExecuteC</condition>
        <!-- Activity C -->
        <else>
            <sequence>
                <!-- Activity B -->
                <if>
                    <condition>$ShouldNotEndLoop</condition>
                    <!-- Duplicate of Activity C -->
                    <else>
                        <!-- set ShouldExecuteLoop to false -->
                    </else>
                </if>
            </sequence>
        </else>
    </if>
<while>
```

WCP-22 Recursion: An activity is able to invoke itself or one of its ancestors in terms of the overall decomposition structure of the process ([38], pp. 44/45).
Solutions: Any process that is accessible as a Web service is in principle able to implement recursion by invoking itself via its Web service interface. However, such a solution violates

a core aspect of this pattern. Recursion has to be implemented by referencing a structure inside the same process model. A Web service invocation is no internal reference, but an external call. Consequently, such a solution does not provide support for this pattern. Another solution is not possible in WF, as it is not possible for an activity to reference itself or an ancestor concerning the overall decomposition structure inside the same process model. Using a `FlowStep` in the flowchart modeling style to point to an ancestor activity in the process model does not count as recursion. In this case, the ancestor relationship does not relate to the decomposition structure, but to normal control-flow dependencies. Interestingly, it is possible to simulate an implementation of the Recursion pattern using custom activities instead of workflow services. It is not permitted to use a custom activity in its own definition. But it is possible to embed a first custom activity as the root activity of a second custom activity. The first activity can in turn make use of the second activity, thereby implementing the Recursion pattern. The resulting workflow is valid concerning the schema definition of XAML. However, this definition seems to violate the intent of its designers as the attempt to compile the workflow results in the livelock of the compiler which seems to be unable to parse the recursive structure. Just as for WF, there is no possibility to reference ancestor activities in WS-BPEL. So there also is no direct support to this pattern.

Termination Patterns:

WCP-11 Implicit Termination: A process instance terminates given there is no more work to perform ([38], p. 25).
Solutions: This pattern resembles the default semantics in both languages. A process instance terminates gracefully in case its flow of control encounters a point where no more activities are defined. The edit distance for both solutions is zero as they are identical to the respective stubs and the solutions provide direct support.

WCP-43 Explicit Termination: A process instance terminates when it reaches a specific state, normally a designated end node. The difference of this pattern to the Cancel Case pattern is that here the process instance should be recorded as having completed successfully ([38], pp. 76/77).
Solutions: In spite of the lack of a designated activity for explicit termination, there is a similar solution in WS-BPEL and WF that achieves the desired effect. This solution is identical to the solutions for the different cancellation patterns found in listing 16 on page 64. Here, the complete process needs to be enclosed by an exception handling mechanism. In case the process should be terminated, simply a specific exception needs to be thrown. This terminates the normal process flow and executes a handler. If this handler is empty, because of implicit termination, the process instance terminates gracefully. For WF, a `TryCatch` activity is needed and the support measures are identical to the ones for the cancellation patterns. For WS-BPEL, the handler can simply be attached to the process `scope` and no additional insertion of a `scope` and `sequence` are required. This provides direct support and reduces the edit distance to six.

Trigger Patterns:

WCP-23 Transient Trigger, WCP-24 Persistent Trigger: An activity can be triggered by a signal from the process or the operating environment. For the Transient Trigger this signal is not durable, meaning it is lost if the activity is not in a state where it can react to it. For

the Persistent Trigger, the signal is durable, meaning it is stored until the activity is in a state where it can react to it ([38], pp. 45 - 49).
Solutions: Triggers in WF and WS-BPEL are implemented by sending messages using activities like the WF `Receive` or the WS-BPEL `onMessage`. In both languages, messages are durable and kept until an activity is able to process it. The process stubs in both languages therefore implement the Persistent Trigger directly with an edit distance of zero. No solutions for the Transient Trigger pattern could be found.

5.3.2 Discussion of Support for Control-flow Patterns

The results of the previous section as well as the comparison to previous analyses can be found in Table 5 on the facing page.

The patterns to which solutions could be found in both, WS-BPEL 2.0 and WF 4, can now be used to compare the degree of selectivity provided by the two support measures. The amount of solutions where a support measure discriminates in relation to the total amount of solutions to the same patterns can be used to quantify the degree of selectivity provided by a support measure. 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, solutions could be found in WF 4. In WS-BPEL 2.0, 31 patterns could be implemented. For Sun BPEL only solutions to 25 patterns could be found. For 29 patterns, solutions could be found in both WF 4 and WS-BPEL 2.0. Only in six cases where solutions could be found, the trivalent measure discriminates for WS-BPEL 2.0 and WF 4. The edit distance discriminates in eighteen of the cases. When comparing WS-BPEL 2.0 and WF 4 with the trivalent measure, the degree of selectivity amounts to $6/29 = 0.21$. When comparing the languages with the edit distance, the number amounts to $18/29 = 0.62$. For all 25 patterns to which solutions could be found in Sun BPEL, also solutions in WF 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 $14/25 = 0.56$. Obviously, the edit distance provides a higher degree of selectivity. The numbers can be found in Table 6 on page 78. Comparing the degree of selectivity between WS-BPEL and Sun BPEL would not make much sense, as the two languages necessarily do not differ strongly. Still, also here the edit distance performs better.

It is not surprising that the degree of support for several patterns, such as Parallel Split or Exclusive Choice, is identical in WF and WS-BPEL even with the edit distance. These patterns relate to concepts that are very well understood and consequently the solutions are very similar. For several patterns, such as the Discriminator and Partial Join patterns in WS-BPEL, 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.

When comparing the results here with those of previous language versions determined in other studies [38, 55], only the trivalent measure is available. It has to be emphasized that here the degree of support was determined using a unified notion for the trivalent measure, as opposed to the previous studies that used criteria specific for each pattern. Therefore in some cases, the degree of support differs although the solutions only changed in the naming of activities.

Table 5: Support of workflow control-flow patterns. If available, the edit distance is shown first followed by the classical trivalent measure in parentheses. A value of '-' for the edit distance means that no valid solution could be found in the scope of the language. 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 [55]	WF 4	WS-BPEL 1.1 taken from [38]	WS-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 (+)

Table 6: Degree of Selectivity of the Support Measures for the Control-flow Patterns

Languages	# pattern solutions in both languages	Trivalent Measure	Edit Distance
WF 4 & WS-BPEL 2.0	28	0.21	0.62
WF 4 & Sun BPEL	25	0.44	0.56

For WS-BPEL much is the same. 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. Other minor changes involve the naming of activities. For instance, serializable `scopes` have been renamed to isolated `scopes`. All in all, WS-BPEL supports a variety of patterns. More efficient solutions to the Discriminator and Partial Join patterns could be implemented by providing a `completionCondition` for the `flow` activity similar to the `forEach` activity. This would enable solutions like in WF. WS-BPEL does not provide support for several patterns due to its structuredness and the inability to create cycles using `links`, as well as its threading model. The lack of `links`, isolated `scopes` and parallel `forEach` activities, severely limits the degree of support provided by Sun BPEL.

When comparing WF 3.5 to WF 4, one thing is obvious: While the solutions of the patterns have changed considerably, 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. The first platform update of .NET 4 (cf. section 2.3) re-introduces this style. So, the platform update might compensate for some of the present deficiencies and increase the degree of pattern support for control-flow patterns provided by WF slightly. 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. This is due to its inability to describe concurrent branches. Obviously, a Parallel Split or Multi-Choice construct is missing in this style.

Altogether, the support provided by WF 4 and WS-BPEL 2.0 is still very similar. A decision against one of the languages in favor of the other one can hardly be based on their support of control-flow structures only. Things are different when deciding between Sun BPEL and WF 4. WF 4 provides support for more patterns and solutions often are also less complex.

5.4 Change Patterns

The change patterns catalog [50,51] describes three different aspects relating to change in PAIS, *adaptation patterns*, *patterns for changes in predefined regions* and *change support features*. Change support features and adaptation patterns describe aspects of the environment used to develop process models and execute process instances that ease their change and maintenance. Change support features are for example a version control system ([51], p. 464) or access control for users performing the changes ([51], p. 467). Adaptation patterns describe operations and functions available in an IDE for developing process models that ensure the consistency of a process model in spite of changes to its control-flow structure. For instance, such operations are the moving of a complete process fragment[19] to another position in the process model ([51], p. 453), or the embedding of a process fragment in a concurrent branch ([51], p. 455). Change support features and adaptation patterns cannot be used to analyze a language, but instead separate components of the development and execution environment of a language. A language such as WS-BPEL, that does not define any development environment, but only the language structure, cannot be analyzed for such patterns. In this study, languages and not development environments are of primary interest. Therefore, change support features and adaptation patterns are excluded from the analysis. Patterns for changes in predefined regions on the other hand do describe constructs in a language similar to the control-flow patterns and therefore fall into the scope of the analysis. The catalog defines four such patterns. Neither WS-BPEL, nor WF have been analyzed for their support for these patterns so far. Still, the authors partly describe how these patterns could be implemented using certain workflow control-flow patterns. This information is used to infer the solutions described below.

All adaptation patterns and also one pattern for changes in predefined regions describe the change of the control-flow structure of a concrete process instance at run-time. Neither WF, nor WS-BPEL or Sun BPEL address the change of the control-flow structure of a single process instance at run-time. In the case of Sun BPEL, to change the structure of a process, the process definition would have to be re-deployed, thereby eliminating all running process instances. In the case of WF, the run-time change of the structure of a process instance is also not possible[20].

5.4.1 Analysis of Patterns for Changes in Predefined Regions

PP-1 Late Selection of Process Fragments: For an activity, only a placeholder is specified during design-time. During run-time, that placeholder is substituted by a concrete implementation ([51], p. 459). Design choices relate to whether the selection of the implementation is

[19] The authors use the notion of a *process fragment* which is largely equivalent to the notion of an activity used here.

[20] Run-time changes to the structure of process instances had been possible in WF 3.5, but are no longer supported in WF 4. Unfortunately, no hint could be found in the documentation of WF 4 concerning this subject. However, several threads on the official WF 4 board relate to this problem. There, Microsoft officials state that WF 4 workflows are not mutable at run-time. Those threads can be found at `http://social.msdn.microsoft.com/Forums/en/wfprerelease/thread/2a3cb7b9-d67a-4b9c-8ea4-881f7658aa8d`, `http://social.msdn.microsoft.com/Forums/en/wfprerelease/thread/25538e1f-9a53-4557-980a-c0f9692f2c4a` or `http://social.msdn.microsoft.com/Forums/en-US/wfprerelease/thread/986e5199-3948-4e4a-8cbe-740cbe4c2c57`.

performed automatically or manually, whether the implementation may be an atomic activity or a sub-process, and whether the selection takes place before or after enabling the placeholder activity.
Solutions: One solution that can be implemented in WF and WS-BPEL is to embed every possible implementation of the placeholder in a *Deferred Choice* construct. This implements a pattern variant with semi-automatic selection. Any atomic activity or sub process can be selected and the selection is performed when enabling the Deferred Choice. The solutions for this pattern are identical to the solutions of the Deferred Choice pattern and can be found in listing 19 on page 67. A minimal solution with two choices provides partial support for WF with an edit distance of nine. WS-BPEL and Sun BPEL achieve direct support with edit distances of eight for WS-BPEL and ten for Sun BPEL.

PP-2 Late Modeling of Process Fragments: For a process fragment, only a placeholder is specified during design-time. During run-time the implementation for the placeholder is modeled on demand by a human modeler for each process instance ([51], pp. 459/460). This requires access to a repository that contains activities which are applicable for late modeling. These activities may be chosen and arranged during late modeling.
Solutions: Obviously, this pattern requires the change of the structure of a process instance. There is no built-in construct in the languages at hand that allows to define such behavior. In fact, none of the languages allows to change control-flow structures at run-time. Consequently, there is no support for this pattern in any of the languages.

PP-3 Late Composition of Process Fragments: For a process fragment, only a placeholder is specified at design-time. At run-time, that placeholder is substituted by a concrete implementation that is composed out of several available process fragments ([51], p. 460). For the Late Selection of Process Fragments pattern, the placeholder is substituted by a single activity or sub-process. Here, it is substituted by a set of activities. This also requires the dynamic definition of the control-flow dependencies between these activities.
Solutions: A solution based on the Deferred Choice pattern as before is not valid for this pattern. Such a solution would require the full specification of all possible execution sequences and the selection of one of them. This would produce valid execution traces for the pattern, but violate its nature of dynamic composition. The flow of control between the activities would not, as demanded, be defined dynamically during the execution of the activity, but be statically fixed. Consequently, the Deferred Choice cannot be used here. The authors state that the *Interleaved Routing* control-flow pattern corresponds to a variant of this pattern. As discussed, WF and WS-BPEL partially support the Interleaved Routing pattern with edit distances of nine and 15 respectively. The solution in Sun BPEL does not provide direct support, but has an edit distance of 40. The solutions can be found in listing 22 on page 71.

PP-4 Multi Instance Activity: Multiple Instances of an activity are executed. The number of required instances is not known at design-time ([51], p. 460). Two control-flow patterns are variants of this pattern, these are *Multiple Instances without A Priori Run-Time Knowledge* and *Multiple Instances with A Priori Run-Time Knowledge*.
Solutions: Neither WF nor WS-BPEL support Multiple Instances without A Priori Run-Time Knowledge, but both do support Multiple Instances with a priori Run-Time Knowledge. The solutions depicted in listing 14 on page 61 provide direct support with an edit distance of seven and six respectively. Sun BPEL on the other hand, does only support Multiple Instances with-

out Synchronization or Multiple Instances with a priori Design-Time Knowledge and therefore does not provide support for this pattern.

5.4.2 Discussion of Support for Patterns for Changes in Predefined Regions

As can be seen in Table 7, WF and WS-BPEL are roughly equivalent concerning their support for patterns for changes in predefined regions. A comparison of the degree of selectivity provided by the support measures is not reasonable here, as the number of patterns is simply too small to provide meaningful results. Nevertheless, it is interesting to see that the edit distance discriminates in all cases, while the trivalent measure does not.

Table 7: Support of Patterns for Changes in Predefined Regions

Pattern	WF 4	WS-BPEL 2.0	Sun BPEL
Patterns for Predefined Changes			
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 (+)	- (-)

All in all, the change patterns are hardly sufficient to analyze languages in isolation. Most of the patterns in this catalog do not address language constructs, but editor operations or the availability of software components in the operating environment of a language. For the few patterns that address language constructs, the analysis here demonstrates that support for control-flow patterns is almost sufficient to also cover these patterns. Still, it can be said that the degree of support provided by Sun BPEL is again considerably limited compared to WS-BPEL or WF.

5.5 Service Interaction Patterns

The service interaction patterns describe interaction scenarios between two or more parties. That is why in almost all cases at least two process models are necessary for realizing a pattern. In general, this involves an initiator process that starts a communication session and a responder process. In several cases, also more than two parties are necessary. Such multi-party scenarios may require more than two process models. Sometimes, also two process models are sufficient for a multi party scenario, because several parties may share the same process definition. Still, at run-time, at least one process instance needs to be executed for each of the parties. In WS-BPEL, it is convenient to use one `partnerLink` for each party a process interacts with. This is independent of whether several `partnerLinks` have identical `partnerLinkTypes`. As discussed in section 4, the edit distance for a solution involving multiple processes is the sum of the edit distances for all process models involved. The trivalent measure for the overall pattern is the minimum degree of support provided by any of the process models.

In their studies [3, 4], the authors discuss possible solutions in WS-BPEL 1.1. Unfortunately, they do not qualify the degree of support with a support measure. Instead, they simply present a description of each pattern, several design choices, and a discussion of whether and how the pattern could be implemented in WS-BPEL 1.1. A comparison with qualified results for WS-BPEL 1.1 like for the control-flow patterns is therefore not possible here. Still, the solutions presented here are constructed based on the descriptions of possible solutions in WS-BPEL 1.1 from [3, 4]. These descriptions vary in detail. They are quite specific for the basic patterns and more open for the more complex patterns, such as the Multi-Responses pattern. The description of a solution in the original sources can be found next to the description of a pattern. Here, the page numbers where a pattern and its solution in WS-BPEL 1.1 is described in [3, 4], can be found in each following pattern summary. The structure of the solutions in WS-BPEL 1.1 will not be discussed explicitly for each pattern. Instead, please refer to the original sources [3, 4]. For WF, there is no corresponding evaluation available, so the solutions to the service interaction patterns in WF are new.

5.5.1 Analysis of Service Interaction Patterns

Single-Transmission Bilateral Interaction Patterns:

SIP-1 Send: A party sends a message to another party. Design choices relate to whether the partners are known at design-time or run-time, whether there is reliable delivery of the message, whether the send is blocking or non-blocking and whether or not the message sent may trigger a fault in response ([4], pp. 4/5).
WF: In WF, there is direct support for this pattern using a single `Send` activity or a `SendReply` activity in case the send is part of a synchronous interaction. The `Send` activity requires the specification of the operation and contract name, endpoint address, binding, and message data. The assignment of the message data to a parameter of the `Send` is part of the parameter definition. Based on the change operations as defined in section 4.2, all these configurations amount to an edit distance of four. The responder process to which the send is directed is identical to a process stub.

WS-BPEL: In WS-BPEL this pattern is directly supported either by using an `invoke` activity without an `outputVariable` or a `reply` activity as a response to a previous `receive` activity in the context of a synchronous interaction. In any case, it requires the configuration of the `partnerLink`, `operation` and `portType` and the setting of the input data by referencing a `variable`. This `variable` needs to be defined in the process model and must be assigned the data to be transmitted in advance to the `invoke` activity. Also, the `partnerLink` must be defined as it is not present in the process stub. Due to these additional changes needed in the process model, the edit distance of the resulting solution is quite high compared to WF. It amounts to a value of seven. Given `partnerLinks` and `variables` are in place, future `invokes` will be less costly. For instance, an `invoke` activity that uses a predefined `partnerLink` and an initialized `variable` has an edit distance of three.

Listing 25: Send Pattern

```
1 <!--WF Solution-->
2 <Send OperationName="PartnerOperation" ServiceContractName="PartnerService">
3     <Send.Endpoint AddressUri="http://address:port/PartnerService.xamlx?wsdl">
4         <Endpoint.Binding>
5             <BasicHttpBinding/>
6         </Endpoint.Binding>
7     </Send.Endpoint>
8     <SendParametersContent>
9         <InArgument Key="Message">[MessageContent]</InArgument>
10    </SendParametersContent>
11 </Send>
12
13 <!--BPEL Solution-->
14 <assign>
15     <!-- assign data to variable Message -->
16 </assign>
17 <invoke partnerLink="PartnerService" operation="PartnerOperation" portType="
    PartnerServicePortType" inputVariable="Message"/>
```

SIP-2 Receive: A party receives a message from another party ([4], pp. 5/6). Design choices again relate to the reliability of the transmission and whether messages are persistent or transient[21] ([4], pp. 5/6).
Solutions: WF directly supports the pattern using a `Receive` activity. Here again `OperationName` and `ServiceContractName` need to be set and optional input can be defined. The activity might also create a new process instance. WS-BPEL offers direct support for this pattern with the `receive` activity or with an `onMessage` activity. Again the `partnerLink`, `operation` and `portType`, as well as optional input data need to be specified. In both cases, the pattern is implemented by the two processes that also implemented the previous pattern. Here, the responder process is of relevance. The support measures are identical. Both languages provide direct support with an edit distance of four for WF and seven for WS-BPEL.

Listing 26: Receive Pattern

```
1 <!--WF Solution-->
2 <Receive CanCreateInstance="True" OperationName="StartProcess" ServiceContractName="
    ResponderProcess">
3     <ReceiveParametersContent>
4         <OutArgument Key="Input">[MessageData]</OutArgument>
5     </ReceiveParametersContent>
```

[21] For this aspect, please refer to the discussion of the *Persistent Trigger* and *Transient Trigger* workflow control-flow patterns on page 75.

```
6 </Receive>
7
8 <!--BPEL Solution-->
9 <receive createInstance="yes" partnerLink="MyPartnerLink" operation="startProcess" portType
    ="MyPortType" variable="MessageData"/>
```

SIP-3 Send/Receive: In a bilateral interaction, a party X sends a message to a party Y. Thereafter, Y responds to X ([4], pp. 6/7). Design choices relate to whether the two parties know each other before the interaction, whether there may be fault messages and whether X blocks after sending the first message or not. In any case, incoming and outgoing messages must be correlated. It must be clear that the second message is the response to the first.
WF: There are several activities that provide this functionality in WF. A straight-forward solution would consist of the combination of two `Send` and `Receive` activities on either side. However, using a combination of the two activity templates `SendAndReceiveReply` and `ReceiveAndSendReply` reduces the edit distance. These templates are part of the BAL and count as normal activities. Although they introduce multiple activities, their insertion counts as a single change operation. Apart from two messaging operations that are configured in the same fashion as for the previous patterns, they also introduce a `CorrelationHandle` and configure it correctly, thus easing the implementation of this pattern. Still, operation and contract names, as well as parameters have to be set. This solution provides direct support, because each of the templates counts as a single activity. The edit distance of the process for partner X is five and of partner Y is four. So in total, it amounts to nine.
WS-BPEL: Also in WS-BPEL, this pattern can either be implemented asynchronously or synchronously. For the synchronous case, no `correlationSets` and a single `invoke` activity with an `inputVariable` and an `outputVariable` is sufficient for party X. Again, additional `partnerLink` and `variable` definitions are necessary. Here, X blocks until it receives the answer from party Y. Y on the other hand, requires a `receive` activity as before and a `reply` activity later on which transmits the response message to X. Also, message data needs to be assigned to a `variable`. The edit distance for this solution is considerably higher than for WF. For the initiator process it amounts to nine and for the responder process to six. So, the overall edit distance is 15. As the responder process requires at least two constructs, the solution provides partial support.

Listing 27: Send/Receive Pattern

```
1 <!--WF Solution Party X-->
2 <Send Name="Send">
3     <Send.CorrelationInitializer>
4         <RequestReplyCorrelationInitializer CorrelationHandle="[CorrelationHandle]"/>
5     </Send.CorrelationInitializer>
6 </Send>
7 <ReceiveReply Request="{Reference Send}"/>
8
9 <!--WF Solution Party Y-->
10 <Receive Name="Receive">
11     <Receive.CorrelationInitializer>
12         <RequestReplyCorrelationInitializer CorrelationHandle="[CorrelationHandle]"/>
13     </Receive.CorrelationInitializer>
14 </Receive>
15 <SendReply Request="{Reference Receive}"/>
16
17 <!--BPEL Implementation Party X-->
18 <assign>
19     <!-- Initialize variable Input-->
```

```
20 </assign>
21 <invoke operation="SyncOperation" inputVariable="Input" outputVariable="Output"/>
22
23 <!--BPEL Implementation Party Y-->
24 <receive operation="SyncOperation" variable="Input"/>
25 <assign>
26     <!-- Assign output variable-->
27 </assign>
28 <reply operation="SyncOperation" variable="Output"/>
```

Single-Transmission Multilateral Interaction Patterns:
As discussed in section 4.3, for all of the following patterns, basic messaging operations such as sends or receives do not influence the trivalent support measure.

SIP-4 Racing Incoming Messages: A party expects to receive one of a set of messages. The messages may differ in their structure and originate from different partners ([3], pp. 304/305). An additional design choice regards the way in which messages are processed in case they are ready for consumption at the same time. It is either possible to make a non-deterministic choice of one of the available messages or to apply a previously defined ranking of messages. This pattern is a variant of the *Deferred Choice* control-flow pattern where the choice is based on the arrival of messages.
Solutions: Both solutions for WF and WS-BPEL of the Deferred Choice pattern found in listing 19 on page 67 are also valid for this pattern. In the case of WS-BPEL, this is a `pick` activity, containing multiple `onMessage` activities. In WF, it is accomplished by a `Pick` activity containing multiple `PickBranches` that have a `Receive` activity as `Trigger`. The WS-BPEL standard states that the selection of a message in the case of simultaneous reception is implementation dependent ([31], p. 100). This is considered to count as a non-deterministic choice. WF and Sun BPEL choose among competing events deterministically. In the case of simultaneous completion of two triggers, WF executes the branch of the trigger that was defined first according to the lexicographical order of the process model. Sun BPEL, on the other hand, executes the branch that is defined last according to the lexicographical order. This implies that it is possible in WF and Sun BPEL to express a ranking of messages through the order of the definition of activities in the process model. Consequently, Sun BPEL, WS-BPEL, and WF provide direct support. The edit distances are equal to the solutions of the Deferred Choice pattern which are eight for WS-BPEL, nine for WF and ten for Sun BPEL.

SIP-5 One-To-Many Send: A party simultaneously sends messages of the same type to several parties ([3], pp. 306/307). The number of parties may be known at design time and reliable delivery may or may not be required.
WF: There is no single activity in WF that achieves the desired result, but the case where the number of parties is known at design time can be captured in a straight-forward way by embedding multiple `Send` activities in a `Parallel` activity. There is no requirement here that the activities performing the sending share the same definition (as for one of the *Multiple Instances* control-flow patterns). They may also be separately defined in the process model. Here, basic messaging constructs do not count to the amount of constructs required, so WF does provide direct support. The responder processes are process stubs and have an edit distance of zero. The initiator process has an edit distance of nine.
WS-BPEL: In WS-BPEL, basically the same as for WF applies. The solution is based on the `invoke` activity in combination with the `flow` or `forEach` activity. Multiple responding parties

are captured by multiple `partnerLinks`. The messages sent need to have the same XML Schema definition. In this case, even the same `variable` can be used as input for multiple requests. Consequently, WS-BPEL provides direct support and the edit distance of the initiator process amounts to twelve.

Listing 28: One-To-Many Send Pattern

```
<!--WF Solution-->
<Parallel>
    <Send ServiceContractName="Responder1">
        <SendParametersContent>
            <InArgument TypeArguments="MessageType" />
        </SendParametersContent>
    </Send>
    <Send ServiceContractName="Responder2">
        <SendParametersContent>
            <InArgument TypeArguments="MessageType" />
        </SendParametersContent>
    </Send>
</Parallel>

<!--BPEL Solution-->
<flow>
    <invoke partnerLink="Responder1" inputVariable="inputMessage"/>
    <invoke partnerLink="Responder2" inputVariable="inputMessage"/>
</flow>
```

SIP-6 One-From-Many Receive: A party receives several logically and timely related messages from different parties which are correlated to a single request ([3], pp. 307–309). To avoid waiting for messages infinitely, a solution should incorporate a timeout. A solution may also stop waiting for the reception of further messages, after some of them have been received.
WF: To produce a working sample, a solution to this pattern must be embedded in a larger conversation. To establish the prerequisite for a One-From-Many-Receive, first a One-To-Many-Send is necessary, where the initiator notifies several responders of the fact that it expects to receive multiple messages. Consequently the processes for this pattern build on the solution for the previous pattern. In WF, to enforce the timeout, a `Pick` activity with two branches is necessary. One branch contains a `Delay` activity with the timeout as trigger. The other branch contains multiple `Receive` activities as trigger. To allow for parallel receives it is necessary to encapsulate these activities in a `Parallel` activity. It is important to note that each of the conversations between the initiator and the different partners requires the definition of a unique `CorrelationHandle`. Executing multiple `Receive` activities in parallel with the same `OperationName`, `ServiceContractName`, and `CorrelationHandle` is not possible. Each of the responder processes extends the stub with a single `Send` activity and thus has an edit distance of four each. In a minimal multilateral example, there are two such responders. The edit distance of the initiator process is considerably larger than for the previous patterns, with the amount of 29. Altogether, this amounts to an edit distance of 37. The pattern can be implemented with two essential constructs on the side of the initiator, the `Pick` and the `Parallel` activity. The solution for the One-To-Many-Send is not essential and does therefore not influence the trivalent measure. This provides partial support.
WS-BPEL: In WS-BPEL a solution can be constructed by using a combination of `flow` and `receive` activities similar to the realization of the One-To-Many-Send pattern. By embedding these activities in a `scope` that is associated with an `onAlarm` activity, timeout constraints are enforced. For correlating messages, a distinct `correlationSet` for each of the `Receives` can

be used. It is important to note that in WS-BPEL, in contrast to WF, it is not possible to execute multiple `receive` activities for the same `partnerLink`, `portType` and `operation` in parallel ([31], pp. 91/92). This is independent of whether the same or distinct `correlationSets` are used by the `receive` activities. Three different faults can be thrown by an engine in such a case, depending on the concrete surroundings. So, if one process instance needs to `receive` several messages in parallel, one of the above attributes of the `receive` activity has to be changed. Here, it is the `operation`. The solution in WS-BPEL requires only one responder process model which is accessed via multiple `partnerLinks` and grants partial support. The responder and initiator processes have an edit distances of 13 and 36. This sums up to an edit distance of 49.

Listing 29: One-From-Many Receive Pattern

```
1 <!--WF Solution Initiator-->
2 <!--Execute One-To-Many Send to transmit notifications and initialize CorrelationHandles-->
3 <Pick>
4     <PickBranch>
5         <PickBranch.Trigger>
6             <Delay Duration="TimeoutDuration"/>
7         </PickBranch.Trigger>
8         <!--Timeout handling logic-->
9     </PickBranch>
10    <PickBranch>
11        <PickBranch.Trigger>
12            <Parallel>
13                <!--multiple Receives with different CorrelationHandles-->
14                <Receive CorrelatesWith="[CorrelationHandle]">
15                        <Receive.CorrelatesOn>
16                            <XPathMessageQuery />
17                        </Receive.CorrelatesOn>
18                </Receive>
19            </Parallel>
20        </PickBranch.Trigger>
21        <!--Logic for conversation success-->
22    </PickBranch>
23 </Pick>
24
25 <!--BPEL Solution Initiator-->
26 <!--Execute One-To-Many Send to transmit notifications and initialize correlationSets-->
27 <scope>
28     <eventHandlers>
29         <onAlarm>
30             <for>"TimeoutDuration"</for>
31             <scope>
32                 <!--Timeout handling logic-->
33             </scope>
34         </onAlarm>
35     </eventHandler>
36     <flow>
37         <!--multiple receives with different correlationSets-->
38         <receive>
39             <correlations>
40                 <correlation set="CorrelationSet1" />
41             </correlations>
42         </receive>
43     </flow>
44     <!--Logic for conversation success-->
45 </scope>
```

SIP-7 One-To-Many Send/Receive: A party sends a request to several other parties which may or may not answer in a given time frame ([3], pp. 309/310). According to the pattern definition, sending and receiving messages takes places at the same time and the communication has to be performed asynchronously, e.g. parallelizing a set of synchronous communication interactions violates a core aspect of the pattern. Instead, send and receive operations must happen in parallel. This means that the solutions for the previous pattern are not valid for this pattern as sending and receiving takes place in two distinct phases there. Each pair of send and receives needs to be correlated together in isolation as it marks a separate conversation. The key problem with this pattern is that it must be possible for the correlation mechanism in a language to initialize and use correlation variables in parallel.
WF: To support this pattern, a solution would have to perform the send and the receive operations in one `Parallel` construct and correlations must be initialized in parallel to their use. Distinct `CorrelationHandles` are needed for each pair of `Send` and `Receive` activities, and the `Send` activities initialize the `CorrelationHandles` as they mark the beginning of a conversation. To provide a valid implementation of the pattern, it must be possible for a `Receive` activity to become active before the corresponding `Send` takes place. Because of the single-threaded model of WF, activities will necessarily become enabled at different times. As demonstrated in the context of the Parallel Split pattern on page 51, the activity that is defined first in the `Parallel` is enabled first. By placing a `Receive` activity before its corresponding `Send` activity, it is ensured that the `Receive` starts executing first. In this case, the `CorrelationHandle` used by the `Receive` is not yet initialized. Because there is no message to be received, the `Receive` activity becomes idle and the control is passed to the next activity in the `Parallel`. That way, ultimately a `Send` activity is enabled that initiates a communication session with a responder and initializes the `CorrelationHandle`. Eventually, this responder sends a reply and the previous `Receive` activity becomes active again. This behavior is valid according to the pattern definition. The necessary timer can be implemented using a `Pick` and a `Delay` activity as before. The solution to this pattern is very similar to the solutions of the previous two patterns found in listings 29 on the previous page and 28 on page 86. Here, also the initial sends are to be performed inside the second `Parallel` activity. All in all, the solution to this pattern is a restructuring of the previous solution, where one `Parallel` activity less is needed. Therefore, the support measures are almost identical. Just the edit distance is one point smaller.
WS-BPEL: In WS-BPEL, the pattern cannot be implemented. Having corresponding `invoke` and `receive` activities execute in different branches of the same `flow` activity is not possible. The `receive` activities could get enabled at the same time as the `invoke` activities and the `correlationSets` used in the `receive` activities would not yet be initialized. This triggers a `correlationViolation` fault ([31], pp. 76/77). However, the concurrent enabling of sends and receives is a core aspect of the pattern. The only possibility for a workaround is to use a global `correlationSet` for all messaging operations. This makes it possible to direct messages to the right process instance, but it does not allow to link send and receive operations with each other. Such a connection could only be established by manually manipulating and looking up values inside the messages. This would require extensive application level code which is out of scope for WS-BPEL here. As the pattern cannot be implemented with the built-in correlation mechanism, WS-BPEL does not support it.

Listing 30: One-To-Many Send/Receive Pattern in WF

```
1 <!--WF Solution Initiator-->
2 <Pick>
3     <PickBranch>
4         <PickBranch.Trigger>
5             <Delay Duration="TimeoutDuration"/>
6         </PickBranch.Trigger>
7         <!--Timeout handling logic-->
8     </PickBranch>
9     <PickBranch>
10        <PickBranch.Trigger>
11            <Parallel>
12                <!--multiple Send/Receive pairs with unique CorrelationHandles-->
13                <Receive CorrelatesWith="[CorrelationHandle]">
14                        <Receive.CorrelatesOn>
15                            <XPathMessageQuery />
16                        </Receive.CorrelatesOn>
17                </Receive>
18                <Send>
19                    <Send.CorrelationInitializer>
20                        <QueryCorrelationInitializer CorrelationHandle="[CorrelationHandle]
                              "/>
21                    </Send.CorrelationInitializer>
22                </Send>
23            </Parallel>
24        </PickBranch.Trigger>
25        <!--Logic for conversation success-->
26    </PickBranch>
27 </Pick>
```

Multi-Transmission Interaction Patterns:

SIP-8 Multi-Responses: A party X sends a request to a party Y. Subsequently, Y sends responses for the request to X until either,

1. X sends a notification to stop.
2. X reaches a deadline for accepting messages.
3. there is an interval of inactivity from Y.
4. Y indicates that no more responses will be sent.

The messages sent may be of different type, so a solution for this pattern needs to incorporate a solution to the Racing Incoming Messages pattern. The number of messages necessary is determined at run-time. In case X decides to stop the interaction there is an interval in which Y may still send responses to X, because it is not yet aware of the ending of the communication session. Therefore, some mechanism needs to be in place that informs Y of the rejection of its messages ([3], pp. 310 - 312). Such a mechanism can be a notification from X to Y. In other words, both X and Y, need to be aware of the ending of the communication session. As demonstrated by the edit distances of the following solutions, this pattern is the most complex pattern implemented during the analysis.
WF: In WF, the initiator process for this pattern can be implemented based on the `Pick` activity. This activity is needed to enforce a global conversation timeout. One of the branches of the `Pick` activity has a `Delay` activity with the global timeout as trigger, the other one contains the messaging logic as trigger. In case the `Delay` activity completes before the other trigger, the global timeout is fired. The other trigger is a looping activity that surrounds one

more Pick activity. The looping activity is necessary to be able to receive multiple messages and must be active as long as further responses are expected. The Pick inside the looping activity contains three PickBranches. One branch expects and processes regular responses from Y. A second branch expects a stop message from Y and terminates the loop if such a message is received. The third branch again contains a timer that checks for a period of inactivity by Y. If one of the timeouts fires, X sends a stop notification to Y. The structure of the responder process is similar to that of the initiator with the exception of the first Pick activity. This activity is not necessary, as the global conversation timeout is only maintained by the initiator. Still, also here, a loop that contains a Pick activity is needed. One of the branches of the Pick needs to decide whether to send a regular response or a notification to stop. Here, these two operations were implemented in two alternative branches. Another branch further waits for a notification from X that indicates that the communication session should stop. Obviously, both processes are rather complex and there is no direct support for this pattern according to the trivalent measure. The edit distance of the initiator process is 41 and that of the responder process is 30, amounting to a total of 71.

WS-BPEL: In WS-BPEL the solution to this pattern is quite similar. The main difference lies in the realization of timeouts in WS-BPEL. Also here, the pattern can be captured by embedding a pick activity with one onMessage for each of the possible message types in a while activity. The triggering of the stop of the interaction on either side can be accomplished by onMessage activities that expect certain stop messages. Periods of inactivity by Y can be controlled by an onAlarm inside the pick. To enforce the global timeout, the loop needs to be embedded in a scope with an onAlarm. The handler contains the transmission of the notification to the responder and an exit activity to terminate the conversation. Altogether, also here the number of constructs required is too high for this solution to provide support for the pattern according to the trivalent measure. The edit distance of the initiator process is 58 and of the responder process 32 which amounts to a total of 90.

Listing 31: Multi-Responses Pattern in WF

```
1 <!--Initiator-->
2 <Send OperationName="SendInitialRequest"/>
3 <Pick>
4     <PickBranch>
5         <PickBranch.Trigger>
6             <Delay Name="GlobalConversationTimeout" />
7         </PickBranch.Trigger>
8         <Send OperationName="NotifyResponderOfStop" />
9     </PickBranch>
10
11     <PickBranch>
12         <PickBranch.Trigger>
13             <While Condition="Not EndConversation">
14                 <Pick>
15                     <PickBranch>
16                         <PickBranch.Trigger>
17                             <Receive OperationName="ReceiveResponse" />
18                         </PickBranch.Trigger>
19                         <!--process response-->
20                     </PickBranch>
21                     <PickBranch>
22                         <PickBranch.Trigger>
23                             <Receive OperationName="ReceiveStopFromResponder" />
24                         </PickBranch.Trigger>
25                         <Assign>
```

```
26                             <!--set EndConversation to true-->
27                         </Assign>
28                     </PickBranch>
29                     <PickBranch>
30                         <PickBranch.Trigger>
31                             <Delay Name="PeriodOfInactivity" />
32                         </PickBranch.Trigger>
33                         <Sequence>
34                             <Send OperationName="NotifyResponderOfStop" />
35                             <Assign>
36                                 <!--set EndConversation to true-->
37                             </Assign>
38                         </Sequence>
39                     </PickBranch>
40                 </Pick>
41             </While>
42     </PickBranch>
43 </Pick>
44
45 <!--Responder-->
46 <Receive OperationName="SendInitialRequest" />
47 <While Condition="[Not Terminate]">
48     <Pick>
49         <PickBranch>
50             <PickBranch.Trigger>
51                 <!--Decide to send regular response-->
52             </PickBranch.Trigger>
53             <Send OperationName="ReceiveResponse">
54         </PickBranch>
55         <PickBranch>
56             <PickBranch.Trigger>
57                 <!--Decide to end conversation-->
58             </PickBranch.Trigger>
59             <Sequence>
60                 <Send OperationName="ReceiveStopFromResponder" />
61                 <Assign>
62                     <!--Set Terminate to True-->
63                 </Assign>
64             </Sequence>
65         </PickBranch>
66         <PickBranch>
67             <PickBranch.Trigger>
68                 <Receive OperationName="NotifyResponderOfStop" />
69             </PickBranch.Trigger>
70             <Assign>
71                 <!--Set Terminate to True-->
72             </Assign>
73         </PickBranch>
74     </Pick>
75 </While>
```

Listing 32: Multi-Responses Pattern in WS-BPEL

```
1 <!--Initiator-->
2 <invoke operation="SendInitialRequest" />
3 <scope>
4     <eventHandlers>
5         <onAlarm>
6             <for>"GlobalTimeout"</for>
7             <scope>
8                 <sequence>
9                     <invoke operation="NotifyResponderOfStop" />
10                    <exit />
11                </sequence>
```

```
12          </scope>
13        </onAlarm>
14    </eventHandlers>
15    <while>
16        <condition>$MoreResponsesNeeded</condition>
17        <pick>
18            <onMessage operation="ReceiveResponse">
19                <!--process response-->
20            </onMessage>
21            <onMessage operation="ReceiveStopFromResponder">
22                <assign>
23                    <!--set MoreResponsesNeeded to false-->
24                </assign>
25            </onMessage>
26            <onAlarm>
27                <for>"PeriodOfInactivity"</for>
28                <sequence>
29                    <invoke operation="NotifyResponderOfStop" />
30                    <assign>
31                        <!--set MoreResponsesNeeded to false-->
32                    </assign>
33                </sequence>
34            </onAlarm>
35        </pick>
36    </while>
37 </scope>
38
39 <!--Responder-->
40 <receive operation="SendInitialRequest" />
41 <repeatUntil>
42    <pick>
43        <onMessage operation="NotifyResponderOfStop">
44            <assign>
45                <!--Set NoMoreResponses to true-->
46            </assign>
47        </onMessage>
48        <onAlarm>
49            <sequence>
50                <assign>
51                    <!--Set message regular data-->
52                </assign>
53                <invoke operation="ReceiveResponse" />
54            </sequence>
55        </onAlarm>
56        <onAlarm>
57            <sequence>
58                <assign>
59                    <!--Set stop message data-->
60                </assign>
61                <invoke operation="ReceiveStopFromResponder" />
62            </sequence>
63        </onAlarm>
64    </pick>
65    <condition>$NoMoreResponses</condition>
66 </repeatUntil>
```

SIP-9 Contingent requests: A party X sends a request to a party Y. If Y does not answer in a given time frame, X sends the request to another party Z ([3], p. 312). An answer from Y that arrives too late may be rejected.
WF: This pattern can again be captured using the `Pick` activity with a `Receive` activity as trigger next to a `Delay` activity as trigger. The initial request is transmitted using a `Send`

activity. The branch with the `Receive` waits for a first response. The branch with the `Delay` enforces the timeout and contains the logic to send the request to another party which can be done using a pair of `Send` and `Receive`. If the first responder answers after the timeout has fired, the initiator is no longer in a state to process the message. This corresponds to a rejection of the message. The only essential construct used is the `pick` activity, so this solution provides direct support. Two rather trivial responder processes are needed. One of them is identical to the process stub and accepts a message without sending any response. The edit distance of this process is zero. The second responder accepts a message and sends a response using a `Send` activity. The edit distance of this process is four. The edit distance of the initiator process is 24, so in total the edit distance of the solution amounts to 28.
WS-BPEL: The structure of the solution in WS-BPEL is identical to the one in WF. The pattern is captured using the `pick` activity with an `onMessage` and an `onAlarm` activity. The `onAlarm` enforces the timeout for the first request and contains the logic to invoke another partner. There are two responders, one that ignores the request and one that sends a response. Again, the initiator process uses two `partnerLinks`. Also, this solution provides direct support. The edit distance of the initiator process is 27 and that of the second responder process is seven, which is 34 in total.

Listing 33: Contingent Requests Pattern

```
1  <!--WF Solution Initiator-->
2  <Send OperationName="SendRequest" ServiceContractName="FirstResponder" />
3  <Pick>
4      <PickBranch>
5          <PickBranch.Trigger>
6              <Receive OperationName="ReceiveResponseFromFirstResponder" />
7          </PickBranch.Trigger>
8          <!--Logic for processing the response-->
9      </PickBranch>
10     <PickBranch>
11         <PickBranch.Trigger>
12             <Delay Duration="TimeoutForFirstResponse" />
13         </PickBranch.Trigger>
14         <Sequence>
15             <Send OperationName="SendRequest" ServiceContractName="SecondResponder" />
16             <!-- Replicate the Pick for each following contingent request-->
17         </Sequence>
18     </PickBranch>
19 </Pick>
20
21 <!--BPEL Solution Initiator-->
22 <invoke operation="SendRequest" partnerLink="FirstResponder" />
23 <pick>
24     <onMessage operation="ReceiveResponseFromFirstResponder">
25         <!--Logic for processing the response-->
26     </onMessage>
27     <onAlarm>
28         <for>"TimeoutForFirstResponse"</for>
29         <sequence>
30             <invoke operation="SendRequest" partnerLink="SecondResponder" />
31             <!-- Replicate the pick for each following contingent request-->
32         </sequence>
33     </onAlarm>
34 </pick>
```

SIP-10 Atomic multicast notification: A party sends notifications to several other parties. A subset of these parties is required to accept the notification in a certain time frame ([3], pp. 312–314). The interaction is divided in two phases. First, a set of notifications is sent from the initiating party to a set of responders. Then, the responders may answer with confirmations to the initiator. The core issue of this pattern is that a solution needs to provide transactional semantics for the conversation. Either the sending of all notifications is successful, or the conversation needs to be rolled back. According to one design choice, the set of parties to which the notification is sent may not be known at design- or run-time which means that this pattern is based on the *Multiple Instances without a priori Run-Time Knowledge* workflow control-flow pattern. Furthermore, a solution should support a way to specify the minimum or maximum amount of parties that need to accept the notification.
WF: WF provides transactional support with the help of the `TransactionScope` and `TransactedReceiveScope` activities. Both of these activities establish transactional semantics, including transaction rollback. A transaction can be started on the side of the initiator with the `TransactionScope`. Inside this scope, the initiator sends the notification to the responders. In case the transmission of one of the notifications fails, the transaction can be rolled back, so there is an all-or-nothing semantics provided for the sending of the notification, as demanded by one of the design choices. Using the notification, a transaction can be triggered on the side of the responders with the `TransactedReceiveScope`. For this, the `Receive` activity that receives the notification needs to be put into the `Request` part of the `TransactedReceiveScope` activity. All processing, as well as the sending of the response is then put into the `Body` of `TransactedReceiveScope`. To ensure that the notifications are sent simultaneously, the sending part on the side of the initiator needs to be embedded in a `Parallel` or `ParallelForEach<T>` activity. The `Receive` activities that wait for the responses also need to be scheduled in parallel. The `CompletionCondition` attributes of these constructs enable a premature stop condition, as well as minimum or maximum bounds of successful responses. Thereby, WF is able to fulfill all but one design choice. As WF does not support the *Multiple Instances without a priori Run-Time Knowledge* pattern (cf. page 61), one of the choices cannot be met. The number of constructs needed on the side of the initiator amounts to three which means that there is no support provided by this solution according to the trivalent measure. The edit distance of the initiator process is 22. At least two responder processes are needed (otherwise there would be no multicast) and each of these processes has an edit distance of nine. All in all, this amounts to an edit distance of 40.
WS-BPEL: WS-BPEL does not provide built-in transactional support and thereby misses the central issue of this pattern ([3], p. 313). There is a way to emulate a *quasi-atomic* transaction using `scopes` and `compensationHandlers` on the side of the respondents. This would enable them to compensate for already completed work, instead of rolling it back. The authors of the pattern state that a quasi-atomic transaction is not sufficient for providing support for this pattern and thus there is also no support by WS-BPEL[22].

Listing 34: Atomic Multicast Notification Pattern in WF

```
1 <!--Initiating Party-->
2 <TransactionScope>
3     <Sequence>
4         <Parallel>
```

[22]This pattern falls into the scope of a WS-* standard, *atomic transaction*, available at `http://docs.oasis-open.org/ws-tx/wsat/2006/06`. However, this is not in the scope of WS-BPEL.

```
5             <!--Multiple Sends for transmitting notifications-->
6         </Parallel>
7         <!--First phase ended-->
8         <Parallel>
9             <!--Multiple Receives for getting responses-->
10        </Parallel>
11    </Sequence>
12 </TransactionScope>
13
14 <!--Responding Party-->
15 <TransactedReceiveScope>
16     <TransactedReceiveScope.Request>
17         <Receive OperationName="ReceiveNotification" />
18     </TransactedReceiveScope.Request>
19     <Send OperationName="SendConfirmation">
20 </TransactedReceiveScope>
```

Routing patterns:
All of the routing patterns involve at least three parties, otherwise, there would not be any routing. Normally, these parties are an initiator, a responder and a third party. For all routing patterns, it must be possible to transmit references, endpoint addresses of services, and dynamically bind them to be used in messaging operations.

SIP-11 Request with referral: Party A sends a request to party B, indicating that any responses should be sent to one or more other parties ([3], pp. 314/315). Faults may be sent to A as well. The central issue of this pattern clearly relates to reference passing and dynamic partner binding. The simple case where B does already have full information about the third parties reduces the problem to the transmission of correlation data from A to B that allows B to communicate with specific process instances of the third parties. For all languages here, this is trivial. To provide a valid solution to this pattern, a language must support reference passing and dynamic binding.
WF: Endpoint addresses can be used like ordinary variables in WF. Endpoint and correlation data can simply be transmitted as other message data. A variable of type `String` that stores an endpoint address is sufficient for this task. Three processes are needed for a minimal solution. The structure of these processes is trivial and consists of several `Send` and `Receive` operations on either side. First the initiator process transmits a request to a second process. This second process takes the role of party A and passes the request and a reference of the initiator process to a third party. The third party uses the reference to send a response to the initiator. As only normal messaging activities are used, the solution provides direct support. The edit distance of the initiator process is ten, that of the second process is five and that of the third party is six. In total this amounts to an edit distance of 21.
WS-BPEL: WSDL endpoint references are the basic means for achieving reference passing and dynamic binding of `partnerLinks` in WS-BPEL. The WS-BPEL 2.0 standard provides *service references* as a container for WSDL endpoint references. `PartnerLinks` can be bound to a service reference at runtime which is done in an ordinary `assign` activity. The service reference can be treated just as a normal `variable`. Thus, it is possible to transmit the service reference data to a process instance via an inbound messaging operation and to assign that data to a `partnerLink`. So, to capture this pattern in WS-BPEL at least three processes are necessary, one of which passes the reference of the second process to the third process which

then invokes the second one[23]. The structure of the three processes is identical to that of the WF solution. Consequently also this solution provides direct support. The edit distance of the initiator process is 13, that of the second process is seven, and that of the third party is eight. In total this amounts to 28.
In Sun BPEL, reference passing is possible. Dynamic partner binding however, could not be accomplished. Erroneous behavior of the engine prevented all scenarios that were tested from working. Consequently, no solution for any of the routing patterns could be found in Sun BPEL. If the reference is read from an incoming message and assigned to a `partnerLink`, a following `invoke` activity using this `partnerLink` fails with an exception in the engine. If a `partnerLink` is overwritten with an endpoint reference that is hard-coded in an `assign` activity of the same process model, the assignment simply has no effect. Although even the BPEL debugger of the Netbeans IDE shows a new endpoint address for the `partnerLink`, subsequent `invoke` activities simply use the old endpoint address. Also, the official examples of the OpenESB project for dynamic binding[24] suffer from this behavior. The WS-BPEL process models provided with this study can easily be modified to demonstrate this behavior by uncommenting certain parts of code in an `assign` activity.

Listing 35: Request with Referral Pattern

```
1 <!--WF Party A-->
2 <Send OperationName="SendRequest" ServiceContractName="PartyB" />
3 <Receive OperationName="ReceiveResponse" />
4
5 <!--WF Party B-->
6 <Receive OperationName="SendRequest" />
7 <Send OperationName="ReceiveReferral" ServiceContractName="ThirdParty" />
8     <SendParametersContent>
9         <InArgument TypeArguments="String">http:/address:port/partyA?wsdl</InArgument>
10    </SendParametersContent>
11 </Send>
12
13 <!--WF ThirdParty-->
14 <Sequence>
15     <Sequence.Variables>
16         <Variable Type="String" Name="ReferredAddressUri"/>
17     </Sequence.Variables>
18     <Receive OperationName="ReceiveReferral">
19         <ReceiveParametersContent>
20             <OutArgument TypeArguments="String">[ReferredAdressUri]</OutArgument>
21         </ReceiveParametersContent>
22     </Receive>
23     <Send EndpointAddress="[New System.Uri(ReferredAddressUri)]" OperationName="
        ReceiveResponse" ServiceContractName="PartyA"/>
24 </Sequence>
25
26 <!--WS-BPEL Party A-->
27 <invoke operation="SendRequest" partnerLink="PartyB" />
28 <receive operation="ReceiveResponse" />
29
30 <!--WS-BPEL Party B-->
31 <receive operation="SendRequest" />
32 <assign>
33     <copy>
```

[23]A comprehensive example for reference passing can also be found in the standard ([31], pp. 184 ff).
[24]The documentation and examples are available at `http://wiki.open-esb.java.net/Wiki.jsp?page=UsingDynamicPartnerLinks` and `http://wiki.open-esb.java.net/Wiki.jsp?page=UsingDynamicPartnerLinksAndDynamicAddressing`

```
34          <from>
35              <literal>
36                  <service-ref>
37                      <EndpointReference>
38                          <!--Endpoint data-->
39                      </EndpointReference>
40                  </service-ref>
41              </literal>
42          </from>
43          <to variable="Referral">
44      </copy>
45  </assign>
46  <invoke operation="ReceiveReferral" variable="Referral" />
47
48  <!--WS-BPEL Third Party-->
49  <variables>
50      <variable name="Referral" element="service-ref"/>
51  </variables>
52  <sequence>
53      <receive operation="ReceiveReferral" inputVariable="Referral"/>
54      <assign>
55          <copy>
56              <from variable="Referral"/>
57              <to partnerLink="PartyA"/>
58          </copy>
59      </assign>
60      <invoke operation="ReceiveResponse" partnerLink="PartyA"/>
61  </sequence>
```

SIP-12 Relayed request: Party A sends a request to party B which delegates the request to a number of third parties. The following interaction takes place between A and the third parties, while party B is observing a view of the interaction ([3], pp. 315/316). This pattern is obviously a specialization of the previous pattern where party B observes a part of the interaction between A and the third parties that takes place after the referral. Therefore, the same considerations as for the previous pattern apply. An additional condition of this pattern is that it must be possible to define views upon messages which should be sent to B. These views are defined at design-time, but may be modified at run-time.
WF: The solution to this pattern is an extension of the previous one, where the third party transmits an additional message to the second party after responding to the initiating party. To incorporate a notion of view with the messages received is rather difficult, because there is no native notion of a message view in WF. It is no problem to send parts of the messages between the initiator and the third party also to the second party, but it is difficult to modify the set of parts which are sent to the second party during run-time. It could be achieved by for example providing several different view definitions at design-time and letting the second party select which one it desires at run-time using a *Deferred Choice* construct. It could also be possible to provide the definition of view fragments at design-time and letting the second party compose the desired view out of these fragments at run-time, again by Deferred Choice. However, these solutions require extensive process-level code and scale very bad for a bigger number of views or configurations. Proper handling of views calls for an additional API that provides view objects. Such an API is not yet part of the BAL. For a minimal example, views can be simple strings. Compared to the solution for the previous pattern, only an additional `Send` activity on the side of the third party and an additional `Receive` activity and a `CorrelationHandle` on the side of the second party is needed. The edit distance of the second party increases to eleven and

that of the third party to ten, that of the initiating party remains the same. Altogether, this amounts to a distance of 31. As the view cannot be modified at run-time, one design choice cannot be fulfilled and the solution provides partial support.
WS-BPEL: The same considerations as for WF also apply to WS-BPEL. Views of messages can be defined in WS-BPEL by extracting information of the messages between the initiator and the third party and assigning this information to messages sent to the second party using the standard assignment mechanism of WS-BPEL which is XPath 1.0 in combination with `assign` activities. More complex views could be constructed using the XML Query Language (XQuery), but this is out of scope for the analysis. Apart from the fact that constructing views with the standard assignment mechanism can be quite complex and that any of the views must be transmitted to the second party using additional messaging operations, this mechanism cannot be modified at run-time. Just as WF, WS-BPEL misses one design choice and therefore achieves partial support. The edit distance of the second process increases to 18 and that of the third party to 16. In total this amounts to 47.

Listing 36: Relayed Request Pattern

```
1 <!--WF Party A-->
2 <Send OperationName="SendRequest" ServiceContractName="PartyB" />
3 <Receive OperationName="ReceiveResponse" />
4
5 <!--WF Party B-->
6 <Receive OperationName="SendRequest" />
7 <Send OperationName="ReceiveReferral" ServiceContractName="ThirdParty" />
8 <Receive OperationName="ReceiveView" />
9
10 <!--WF ThirdParty-->
11 <Receive OperationName="ReceiveReferral" />
12 <Send OperationName="ReceiveResponse" ServiceContractName="PartyA" />
13 <Send OperationName="ReceiveView" ServiceContractName="PartyB" />
14
15 <!--WS-BPEL Party A-->
16 <invoke operation="SendRequest" partnerLink="PartyB" />
17 <receive operation="ReceiveResponse" />
18
19 <!--WS-BPEL Party B-->
20 <receive operation="SendRequest" />
21 <assign>
22     <!--Set referral data-->
23 </assign>
24 <invoke operation="ReceiveReferral" />
25 <receive operation="ReceiveView" />
26
27 <!--WS-BPEL Third Party-->
28 <receive operation="ReceiveReferral" inputVariable="Referral"/>
29 <assign>
30     <!--Set response and view data-->
31 </assign>
32 <invoke operation="ReceiveResponse" partnerLink="PartyA"/>
33 <invoke operation="ReceiveView" partnerLink="PartyB" />
```

SIP-13 Dynamic routing: A request must be routed to several parties based on routing conditions. The order of routing needs to be flexible ([3], pp. 316/317). Apart from dynamic partner binding this pattern requires support for parallelism, support for the *Interleaved Parallel Routing* workflow control-flow pattern, a mechanism for controlling read and write access concerning the documents transmitted and a way for managing role permissions concerning

those who may modify the order of routing.
Solutions: Obviously, the requirements of this pattern are by far out of scope for both languages. Read and write access for documents and role permissions could be implemented using WS-* standards, such as for example WS-Security[25] for controlling read and write access to documents via encryption and signatures. WS-Routing[26] and WS-Referral[27] are proprietary specifications by Microsoft which target to provide parts of the functionality from this pattern, but they have not yet been standardized and it is unlikely that this will happen in the near future. WF also does not support the Interleaved Parallel Routing pattern.

5.5.2 Discussion of Support for Service Interaction Patterns

Table 8: Degree of Selectivity of the Support Measures for the Service Interaction Patterns

Languages	# pattern solutions in both languages	Trivalent Measure	Edit Distance
WF 4 & WS-BPEL 2.0	10	0.2	1
WF 4 & Sun BPEL	8	0.13	1

The results of the previous section are outlined in Table 9 on the next page. For this pattern catalog, again enough patterns are available to compare the degree of selectivity provided by the two support measures in a meaningful way. Ten patterns have solutions in both WS-BPEL and WF. The trivalent measure discriminates only in two cases, so the degree of selectivity provided here is $2/10 = 0.2$. The edit distance discriminates in all cases, so the degree of selectivity provided is 1. For WF and Sun BPEL, common solutions for eight patterns could be found. Only in one case, the trivalent measure discriminates, so degree of selectivity amounts to 0.13. The edit distance again discriminates in all cases. All in all, for this pattern catalog the use of the edit distance is clearly beneficial.

The degree of support for the service interaction patterns provided by WF is considerably better than that of WS-BPEL. Almost all WF solutions are less complex and more patterns are supported. These results can motivate a decision for using WF instead of WS-BPEL. As before, Sun BPEL falls behind the other two languages. The lack of support for dynamic partner binding, resulting from the inability to re-assign endpoint references to `partnerLinks`, is critical.

[25]The documentation of the standard is available at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.
[26]The documentation of the specification is available at `http://msdn.microsoft.com/de-de/library/ms951272%28en-us%29.aspx`.
[27]The documentation of the specification is available at `http://msdn.microsoft.com/de-de/library/ms951244%28en-us%29.aspx`.

Table 9: Support of Service Interaction Patterns

Pattern	WF 4	WS-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	- (-)	- (-)	- (-)

5.6 Time Patterns

There are three types of design choices relevant for the patterns of this catalog [23]. These are *general design choices*, *general design choices for a pattern category* and *pattern-specific design choices* ([23], p. 97). Naturally, a solution must consider all design choices that are relevant to it. The general design choices relevant to all patterns are,

- at what point in time parameters of a pattern are set. This can be at *design-time*, at *instantiation time* of a process instance or at *run-time*.
- at what level of granularity time parameters can be specified. This can be *basic*, being in years, months, minutes, etc., *system-defined*, for example as business days, or *user-defined*, being for example Wednesday afternoon.
- to what process elements a pattern can be applied. These can be *single activities*, including multiple instance activities, *activity sets*, the complete *process instance* or a *set of process instances*.

In the case of WF, there are two activities in the BAL that allow for the specification of time constraints, the `Delay` and the `TransactionScope` activity. Durations are specified as objects of type `TimeSpan`. They can be constructed either by directly using this class or by using one of the other classes of the class library that rely on `TimeSpan` and are able to produce an instance of it, such as `DateTime`. These classes serve as data types for time related aspects. The duration of the time-based activities can be set using variables and activity parameters and thus can be fixed at any time, i.e., at build-time, instantiation time or run-time. So, the first general design

choice will always be fulfilled. Relating to the second design choice, there is limited support for system- or user-defined time parameters, as for example weekdays are available with the `DateTime` class. Basic time parameters are always available, so this design choice will also always be fulfilled. The third general design choice needs to be discussed separately for each pattern.

Time constraints in WS-BPEL are enforced either by using the `wait` activity or an `onAlarm` attached to a `scope` or `pick` activity. To express time constraints, WS-BPEL relies on the basic data types which are defined in the XSD root schema. Durations are expressed using `xsd:duration` and deadlines are expressed either using `xsd:date` or `xsd:dateTime`. To manipulate these data types, WS-BPEL allows for the use of XPath 1.0 expressions [45]. However, XPath 1.0 is not XML Schema aware and therefore none of its built-in functions are capable of producing or manipulating `dateTime` or `date` values[28]. Therefore, these data types are represented as string literals. An engine needs to be able to convert this lexical representation into a valid `date` or `dateTime` at run-time ([31], pp. 58/59). This unfortunately makes the manipulation of dates on process-level rather inconvenient, as it is necessary to cast parts of the string representation to other data types to be able to perform computations and possibly cast the results back to the string representation. As discussed in section 4.1, solutions that require such casting are not valid and therefore not sufficient to provide support for a pattern. There is yet another severe restriction for the time-related capabilities of WS-BPEL. The standard does not define any construct that allows a process instance to access the current system time. Also, XPath 1.0 does not provide such a function. This means that for WS-BPEL, either all dates and durations have to be fixed when defining a process model or have to be supplied through messages sent to the process. Neither one of these solutions are sufficient for providing support for several of the following patterns. For some patterns, the first one is too static and the second one cannot be used, as essential aspects of the pattern would lie outside of the influence of the process instance. It is impossible for a process instance to determine on its own how much time passed between an externally supplied date and the current time at which it is running. The use of an expression language that provides a function for determining the current system time, such as XPath 2.0 [49], would relieve the problem. As discussed in section 5.1.2, in WS-BPEL only the support of XPath 1.0 as expression language is required. Therefore, XPath 2.0 cannot be used here.

Sun BPEL provides several XPath extensions. This includes functions for retrieving the current date and time and comparing string literals that represent dates or durations. Still, Sun BPEL does not provide functions to perform computations based on dates[29]. While it is possible to determine for example that date A lies before date B, it can still not be determined whether their distance is exactly five seconds. For supporting several time patterns, such computations are necessary.

Time-related activities, such as `Delay` in WF or `wait` in WS-BPEL, are essential for supporting time patterns. Therefore in the following, they also count to the trivalent support measure.

[28] XPath 1.0 functions are limited to node set, string, boolean or number functions. They are defined in `http://www.w3.org/TR/xpath/#corelib`.

[29] Such functions are available in XPath 2.0. For a full documentation of these functions, please refer to `http://www.w3.org/TR/xpath-functions/`.

5.6.1 Analysis of Time Patterns

Duration and Time Lag Patterns:
There is one general design choice for the patterns of this category. It relates to whether the duration or time lag is specified as *minimum value*, *maximum value* or *time interval* ([23], p. 98).

TP-1 Time Lags between two Activities: There is a time lag between two activities. This time lag may not only exist between adjacent activities, but between arbitrary ones ([23], pp. 99/100). Due to unclear semantics in case one of the activities is embedded in a loop and the other one is not, the pattern excludes this case. Time lags between the activities can be specified either depending on the start or end of an activity, resulting in four valid combinations for this pattern, being a *Start-Start* relation, a *Start-End* relation, an *End-Start* relation or an *End-End* relation.
WF: The most basic and direct solution for time lags between two succeeding activities is putting a `Delay` activity between them. In this case, the delay is specified as a minimum value and an End-Start relationship. This works for any activities in a process instance. However, these activities must directly succeed each other and this restriction violates a core aspect of the pattern. By leveraging the `DateTime` class, also time lags between arbitrary activities can be expressed. For this, at the time a certain event such as the start or end of an activity occurs, the current system time is assigned to a variable using `DateTime.Now`. Just before the second activity, the time that has passed can be computed by subtracting the previous system time from `DateTime.Now`. In the case of a minimum delay, the delay that is still required needs to be calculated. This can be done by comparing the minimum delay duration to the delay occurred so far. If there is still a delay needed, a `Delay` activity can be executed with the respective duration value[30]. Otherwise, simply the second activity can be executed. To make this decision, an `If` activity is necessary. Maximum delays between arbitrary activities can be implemented in the same fashion. Here, an `If` activity is needed to determine whether too much time has passed, and if this is the case, the necessary exception handling logic can be invoked. Altogether, a pattern variant with only one construct can be implemented. The maximum time lag does not require a `Delay` activity, only an `If` activity. Thereby, WF achieves direct support for the pattern. The edit distance of this solution is eight.
WS-BPEL: To provide support for this pattern, it is necessary to compute the time that has passed between the execution of the two activities and this is not directly achievable neither in WS-BPEL, nor in Sun BPEL. It is possible however, to implement a minimum delay between two succeeding activities by placing a `wait` activity in between them that delays the execution for a certain amount of time or until a fixed date. A maximum delay between two activities could be implemented by embedding the complete control-flow that occurs between them into a `scope` with an `onAlarm` handler. This may enable a maximum delay between connected activities, but still not between arbitrary ones, as the control-flow between them must be structured and may have only a single entry and exit point. If the activities occur in different branches of the process graph, this is not possible. Since the arbitrary positioning of the activities is not a design choice, but a core issue of the pattern, neither WS-BPEL nor Sun BPEL are able to support it.

[30] As a side note, it must be ensured that a `Delay` activity is not executed with a negative duration. `Delay` activities are not able to handle a negative `TimeSpan` and reach a deadlock when confronted with one.

Listing 37: Maximum Time Lags between two Activities in WF

```
<!--Write current time to variable-->
<Assign>
  <Assign.To>
     <OutArgument TypeArguments="DateTime">[ActivityEnd]</OutArgument>
  <Assign.To>
  <Assign.Value>
    <InArgument TypeArguments="DateTime">[DateTime.Now]</InArgument>
  </Assign.Value>
</Assign>
<!--Execute intermediate activities and compute time that has passed-->
<Assign>
    <Assign.To>
        <OutArgument TypeArguments="TimeSpan">[Duration]</OutArgument>
    </Assign.To>
    <Assign.Value>
        <InArgument TypeArguments="TimeSpan">[DateTime.Now - ActivityEnd]</InArgument>
    </Assign.Value>
</Assign>
<If Condition="[TimeSpan.Compare(DelayDuration, MaximumDelay) > 0]">
    <If.Then>
        <!--Trigger exception handling-->
    </If.Then>
</If>
```

TP-2 Durations: A process element[31] has a duration associated with it ([23], p. 100).
WF: Maximum bounds of the execution of an activity or sets of activities can be enforced by embedding them in a `Pick` activity. Two branches are necessary. One branch contains a `Delay` activity as trigger that implements the maximum duration. The other branch contains the activity as trigger with which the duration should be associated. If the `Delay` activity completes first, exception handling can be started. Otherwise, the control-flow can continue normally. The solution requires a `Pick` and a `Delay` activity, therefore it grants partial support. It provides an edit distance of six.
WS-BPEL: This pattern can be implemented in WS-BPEL by embedding the desired process element in a `scope` associated with an `onAlarm` that specifies the maximum duration. Some exception handling in the body of the `onAlarm` is inevitable, otherwise the occurence of the time event would not affect the execution of the activity in the `scope`. One option is to throw a fault which immediately aborts the execution of the body of the `scope`. Minimum durations cannot so easily be enforced, as they again would require the dynamic computation of the remaining amount of delay, based on the amount of time that has passed so far.
Another restriction of Sun BPEL was uncovered during the implementation of this pattern. According to standard WS-BPEL, `for` or `until` elements capture a duration or deadline expression ([31], pp. 58/59). In Sun BPEL, they may only contain string literals and no expressions, even if these expressions return string literals. Consequently, dynamic computation of durations and deadlines, for example using the XPath `concat` function, cannot be implemented. Still, string literals marking durations and deadlines can be read from incoming messages. Such a static solution is still valid for this pattern, so Sun BPEL achieves support. The solution for the maximum duration qualifies for partial support support with an edit distance of seven for WS-BPEL and Sun BPEL.

[31] In this pattern catalog, the authors use the notion of a *process element* which is equivalent to the notion of an activity used here.

Listing 38: Durations Pattern

```
<!--WF Implementation-->
<Pick>
    <PickBranch>
        <PickBranch.Trigger>
            <Delay Duration="MaximumDuration" />
        </PickBranch.Trigger>
    <!--Exception handling logic-->
    <PickBranch>
        <PickBranch.Trigger>
            <!--Activity with associated duration-->
        </PickBranch.Trigger>
    </PickBranch>
</Pick>

<!--BPEL Implementation-->
<scope>
    <eventHandlers>
        <onAlarm>
            <for>"MaximumDuration"</for>
            <throw />
        </onAlarm>
    </eventHandlers>
    <!--Activity with associated duration-->
</scope>
```

TP-3 Time Lags between arbitrary Events: There is a time lag between events. As opposed to *Time Lags between two Activities* pattern, these events might be arbitrary points in a process and not only the start or end of an activity([23], pp. 100/101).
WF: The solution presented for the Time Lags between Two Activities pattern is also a solution for this pattern. The current time can be checked at any point in the process, not only at the starting or ending of activities. So as before, there is direct support for this pattern with an edit distance of eight. The solution can also be found in listing 37.

WS-BPEL: As for the Time Lags between two Activities pattern, a solution for this pattern would require the computation of the time distance between two events dynamically at run-time. As discussed before, neither WS-BPEL nor Sun BPEL offer this capability with built-in features.

Restrictions of Process Execution Point Patterns:
This category comprises patterns that restrict the execution of activities according to certain dates. There is one general design choice for this category. This design choice relates to what kind of date-based restriction applies for a process element. This can either be the earliest or the latest start or completion date.

TP-4 Fixed Date Element: It is possible to determine the latest or earliest time at which an activity is executable based on a fixed date ([23], pp. 101/102). This can be used for realizing deadlines. Such a deadline might differ for each process instance and may be supplied to the instance during run-time or it might be computed by the instance based on a rule encoded in the process model.
WF: There is a simple solution to this pattern that relies on the `DateTime` class and the `Delay` activity. To implement an earliest start date, the delay duration that is still required can be computed by substracting `DateTime.Now` from a fixed date. The fixed date can be supplied in

an incoming message. This solution has an edit distance of three and provides direct support.
WS-BPEL: As the date can be supplied to the process instance, it is possible in WS-BPEL to transmit it via a message. For an earliest start date, the transmitted date must be assigned to the `until` part of a `wait` activity directly before the respective activity. For a latest completion date, the transmitted date must be assigned to the `until` part of the `onAlarm` handler of a `scope` enclosing the respective activity. The solution for an earliest start date qualifies for direct support with an edit distance of three.

Listing 39: Fixed Date Element Pattern

```
1 <!--WF Solution-->
2 <Sequence>
3     <Receive>
4         <ReceiveParametersContent>
5             <OutArgument TypeArguments="DateTime">[FixedDate]</OutArgument>
6         </ReceiveParametersContent>
7     </Receive>
8     <Delay Duration="[FixedDate - DateTime.Now]" />
9 </Sequence>
10
11 <!--BPEL Solution-->
12 <receive variable="EarliestStartDate" />
13 <wait>
14     <until>$EarliestStartDate</until>
15 </wait>
```

TP-5 Schedule Restricted Element: The execution of an activity is restricted according to a schedule, being for example a timetable ([23], pp. 102/103). A schedule is a set of time points or intervals at which an activity or process is executable and depends on a rule to determine the time points based on the current time. The structure of the schedule, e. g. Monday to Friday from 9:00 am to 17:00 pm, is known at design-time. Concrete dates need to be determined by a process instance.
WF: This pattern can be implemented based on the `If` activity and the `DateTime` class. The condition of the `If` activity needs to evaluate to true if `DateTime.Now` is a valid date concerning the predefined schedule. The schedule can be represented by encoding all valid time points in the condition. The solution provides direct support with an edit distance of three and is depicted in listing 40 on the following page.
WS-BPEL: Sending all dates of the schedule to a process as input data is not a valid solution for this pattern. To implement this pattern, a schedule rule is needed which can be used to compare the current date to valid dates. As discussed, the current date or time cannot be obtained in the scope of WS-BPEL, so there is no support for this pattern. As for Sun BPEL, it would be possible to implement a schedule rule purely based on comparison operations, given a measure to represent relative times, such as 18:00 every day. This is not possible with the XSD basic types, as these represent absolute dates. It is, however, possible to retrieve the current date and time using the `current-time()` function and to extract the current hour using `substring()`. By casting this hour string to a number with `number()`, it is possible to check that the current time of the day is before, say 18:00. Unfortunately, this solution still requires data parsing and therefore does not qualify as valid.

Listing 40: Schedule Restricted Element Pattern in WF

```
1 <If Condition="DateTimeNowIsValid">
2     <If.Then>
3         <!-- Perform schedule activity -->
4     </If.Then>
5 </If>
```

TP-6 Time Based Restrictions: The number of times a process element can be executed in a certain time frame is restricted ([23], p. 103). The time frame is characterized either by two events or by a schedule. According to a design choice, the time based restrictions can apply for the activities of a single process instance, but also for activities of all process instances or even for groups of processes. The restrictions may be expressed either for concurrent executions where time frames overlap or by a period of time.
WF: For this pattern, obviously multiple instances of an activity must be executable. This does not necessarily need to be one of the *Multiple Instances* workflow control-flow patterns. The authors relieve the constraint that the multiple instances must execute concurrently. Instead, they can be started and executed in a strictly sequential fashion. Thus, a normal looping construct, such as a `While` activity that contains the restricted activity as body is fully sufficient to implement a variant of this pattern. The `Condition` of the `While` then needs to involve two aspects. Firstly, the number of times the iteration should be performed per se and secondly whether the number of iterations performed does not exceed the predefined limit in the given time frame. The restrictions can be of the same nature as for the previous two patterns. One construct is sufficient, so there is direct support for the pattern with an edit distance of six.
WS-BPEL: As for the previous pattern, this pattern requires access to the current time of execution. So there is no support in WS-BPEL.

Listing 41: Time Based Restrictions Pattern in WF

```
1 <Sequence>
2     <Sequence.Variables>
3         <Variable TypeArguments="DateTime" Default="[DateTime.Now.Add(TimeSpan.FromSeconds
              (5))]" Name="Deadline" />
4         <Variable TypeArguments="Int32" Name="NumberOfDesiredExecutions" />
5         <Variable TypeArguments="Int32" Name="NumberOfPermittedExecutionsPerPeriod"/>
6     </Sequence.Variables>
7     <While>
8         <While.Variables>
9             <Variable TypeArguments="Int32" Default="0" name="Counter" />
10        </While.Variables>
11        <While.Condition>[(Counter &lt; NumberOfDesiredExecutions) And (DateTime.Compare(
              Deadline, DateTime.Now) > 0) And Counter &lt;
              NumberOfPermittedExecutionsPerPeriod]</While.Condition>
12        <Sequence>
13            <Assign>
14                <!--Increment Counter-->
15            </Assign
16        </Sequence>
17    </While>
18 </Sequence>
```

TP-7 Validity Period: The lifetime of a process element is restricted to a given period ([23], p. 103). There are different versions of a process element only one of which is valid at a given time. Once a process element becomes invalid, it cannot be executed anymore. The purpose of this pattern is to aid the evolution of a process model. The process model contains branches

that are valid only before a given date and branches that are valid after this date. When the date is reached, the previous branches become obsolete and may never be executed again. The validity date may be statically encoded in the process model.
WF: Again, this pattern is directly supported by using a single `If` activity or a `Switch<T>` activity, in case multiple validity periods are required. The periods can be determined by fixing the dates that serve as their borders as variables in the process definition. The solution is similar to the solution for the Schedule Restricted Element pattern found in listing 40 on the preceding page with the exception that the fixed date is not dynamically computed, but encoded in the condition of the `If` activity. Furthermore, an `Else` case is necessary for the implementation of the process element that is valid after the date. The edit distance of this solution is four.
WS-BPEL: This pattern requires the access of the current date of execution and the comparison of that date with the fixed validity dates. This is not possible for WS-BPEL. Sun BPEL on the other hand provides exactly the functionality needed with its XPath extensions. Thereby, it provides direct support for this pattern with a solution similar to the one in WF. The edit distance of this solution is also four.

Listing 42: Validity Period Pattern in Sun BPEL

```
1 <if>
2     <condition>date-less-than(current-time(),'ValidityDate')</condition>
3     <!--Execute activity valid before the date-->
4     <else>
5         <!--Execute activity valid after the date-->
6     </else>
7 </if>
```

Variability Patterns:

TP-8 Time Dependent Variability: The control-flow of a process may vary depending on time aspects ([22], p. 9). Aspects considered may be the execution time of an activity or time lags between activities or events. A solution for the *Deferred Choice* workflow control-flow pattern based on time-related triggers is also sufficient to support this pattern.
WF: Any of the solutions that selects one among a set of possible branches based on time-related aspects is valid for this pattern. One of the cheapest solutions in WF is the solution to the Schedule Restricted Element pattern. It selects a control-flow branch using an `If` activity and offers direct support with an edit distance of three. The solution can be found in listing 40 on the facing page.
WS-BPEL: As a solution may be based on the Deferred Choice pattern, also WS-BPEL is able to provide support. This solution consists of an `onMessage` activity and an `onAlarm` activity in a `pick` activity. As it involves a timer, the solution may not be the start of a process instance and therefore also requires the use of `correlationSets` in its messaging operations. This solution provides partial support with an edit distance of eleven. It essentially is the solution to the `Milestone` pattern, as discussed on page 69. In Sun BPEL, the cheapest solution for a pattern that selects one among a set of branches is that of the Validity Period pattern found in listing 42. Consequently Sun BPEL achieves a rating of direct support and an edit distance of four.

Listing 43: Time Dependent Variability in WS-BPEL

```
1 <pick>
2     <onMessage>
3         <!--Execute first branch-->
4     </onMessage>
5     <onAlarm>
6         <!--Execute second branch-->
7     </onAlarm>
8 </pick>
```

Recurrent Process Element Patterns:
There is one general design choice for the two patterns in this category. The number of re-occurrences of process elements may be fixed or dynamic, depend on a time lag and end date, or depend on an exit condition ([22], p. 14).

TP-9 Cyclic Elements: Specific process elements are executed iteratively with time lags in between them ([22], p. 10). This pattern is a specialization of the Time Lags between arbitrary Events pattern. The events between which the delay is defined is the end event of the i-th execution of a process element and the start event of the $(i + 1)$-th execution of the same process element. An example of this pattern is a loop containing two activities, one of which has a time constraint defined on it. Concerning the delay, also the general design choices of the *Durations and Time Lags* category apply (maximum or minimum delays, or a combination of both) and it may be fixed or vary for each iteration.
WF: A solution for this pattern in WF requires at least two constructs. Firstly, a looping construct to control the iterative execution of the activity is needed. Secondly, an additional activity inside the loop is needed to maintain the time constraints of the execution of the first activity. Similar to the solutions for the patterns of the *Durations and Time Lags* category, this needs to be a `Delay` activity enclosed in an `If` activity in case the delay is a minimum value or a single `If` activity in case the delay is maximum value. The solution for the maximum delay uses only two constructs and thereby is able to achieve partial support for the pattern. The edit distance of this solution is twelve.
WS-BPEL: The delay in this pattern is a delay between events that are not directly connected. The events may relate to the same activity, but in between the i-th and $(i + 1)$-th execution of the activity, an arbitrary amount of other activities may be executed and this may take an arbitrary amount of time. Consequently, the delay needed, no matter whether it is a minimum or maximum value, needs to be calculated dynamically, based on the current time. As discussed before, this is not possible, neither in WS-BPEL nor in Sun BPEL.

Listing 44: Cyclic Elements Pattern in WF

```
1 <DoWhile>
2     <DoWhile.Variables>
3         <Variable TypeArguments="DateTime" Default="[DateTime.Now]" Name="LastEndEvent" />
4     </DoWhile.Variables>
5     <Sequence>
6         <If Condition="[DateTime.Compare(DateTime.Now, LastEndEvent.Add(MaximumDelay)) > 0]
              ">
7         <If.Then>
8             <!--Maximum delay exceeded. Trigger exception handling-->
9         </If.Then>
10        <If.Else>
11            <Sequence>
12                <!--Execute cyclic activity-->
```

```
13                 <Assign>
14                     <!--Set LastEndEvent to DateTime.Now-->
15                 </Assign>
16             </Sequence>
17         </If.Else>
18 </DoWhile>
```

TP-10 Periodicity: An activity is executed periodically, based on a periodicity rule ([22], p. 10). This rule may contain one or more dates. A periodicity rule might for example be: Every Monday at 11:30 am. Although this is not contained in the pattern description, it is assumed here that a periodicity rule should involve termination. A valid solution should reach a point where the periodic execution stops and the process instance terminates. Otherwise, the behavior of the instance would resemble a live lock which is not generally a desired state for a process.
WF: To enable repetitive execution of an activity and fulfill the additional assumption, this activity again must be embedded in a looping structure, say, a `DoWhile` activity, that eventually terminates. Then, at the start of this looping structure, a `Delay` activity is needed. The duration of the `Delay` needs to be computed dynamically and be equal to the amount of time between the current time and the next time the activity should be executed. In other words, the periodicity rule is encoded in the computation of the duration of the `Delay` activity. As for the previous pattern, there are two constructs required. So, there is partial support for the pattern. The edit distance amounts to eight.
WS-BPEL: Although it does not provide direct support, there is a rather straight-forward solution in WS-BPEL. Instead of a `for` or `until` element, an `onAlarm` can also contain a `repeatEvery` element[32], the name of which already sounds like an implementation of this pattern. The `repeatEvery` element takes a `duration` and executes the activity contained in the body of the `onAlarm` every time the `duration` expires for as long as the `scope` to which it is attached is active. So, a solution to this pattern consists of putting the activity desired for periodic execution in the body of the handler with a `repeatEvery` element. To enable multiple executions, it has to be ensured that the `scope` is executing for some time which can be done by embedding a `wait` activity in it that specifies for how long, or alternatively until when, the periodic execution should be performed. Altogether, three constructs are needed, the `scope`, the `wait` and the `onAlarm`, so there is no direct support for the pattern. The edit distance of this solution, however, is rather small and equals to seven.

Listing 45: Periodicity Pattern

```
1 <!--WF Solution-->
2 <DoWhile>
3     <Sequence>
4         <Delay Duration="TimeUntilNextExecutionDate" />
5         <!--Execute period activity-->
6     </Sequence>
7 </DoWhile>
8
9 <!--BPEL Solution-->
10 <scope>
11     <eventHandlers>
12         <onAlarm>
13             <repeatEvery>"TimeLagBetweenPeriodActivities"'</repeatEvery>
```

[32] `For` and `until` can also be used in conjunction with `repeatEvery`, but then their semantics differ. For more detailed information please refer to [31], p. 141.

```
14            <scope>
15                <!--Execute period activity-->
16            </scope>
17        </onAlarm>
18    </eventHandlers>
19    <wait>
20        <until>"EndDateOfPeriodicExecution"</until>
21    </wait>
22 </scope>
```

5.6.2 Discussion of Support for Time Patterns

The results of the previous section are outlined in Table 10 on the facing page. As only four patterns are simultaneously supported by WF and WS-BPEL and only five patterns by WF and Sun BPEL, measures for the degree of selectivity are not computed. It can be seen from Table 10 that the edit distance once more provides a higher degree of selectivity.

It is clear from the analysis that the support for time patterns relies heavily on the representation for dates and times used in a language. WF uses sophisticated data types from the .NET class library. Therefore it provides an exceptionally high degree of support for time patterns. Almost all patterns are directly supported and the edit distances are relatively small. As can be seen from the analysis, three aspects must be present in an orchestration language for providing this level of support for time patterns:

1. The availability of basic control-flow structures. Also time-based precedence relationships are based on structural precedence relationships of the process graph. For instance, an activity that should execute before another one concerning the time perspective also has to be placed before that activity in the process graph.
2. The ability of a process instance to access the current system time during its execution. Otherwise, the computation of required delays and enforcement of several constraints is not possible.
3. The availability of data types for dates, times and durations and the possibility to perform arithmetic operations based on these types, like adding five seconds to a given date. Otherwise, the computation of required delays is not possible.

WS-BPEL suffers from deficiencies in the last two aspects. This is due to the fact that WS-BPEL requires only the support for XPath 1.0 as expression language, which completely lacks time-based operations. As the results for Sun BPEL show, already the incorporation of some time-based functions of XPath 2.0 allows to increase the degree of pattern support. In fact, this is the first pattern catalog where Sun BPEL extensions come into play and Sun BPEL excels WS-BPEL. By consolidating the WS-BPEL standard to also require the support for XPath 2.0 as expression language, WS-BPEL would achieve a similar degree of support like WF.

Table 10: Support of Time Patterns

Pattern	WF 4	WS-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 (-)

5.7 Summary

The detailed results for all pattern catalogs can be found in Tables 5, 7, 9, and 10. When looking at the discussions for all pattern catalogs, it is obvious that the edit distance provides a higher degree of selectivity than the trivalent measure. Tables 6 and 8 for the control-flow and service interaction patterns demonstrate this. For the change and time patterns, the case numbers are too small to compute meaningful selectivity measures, but also here it can be seen directly that the edit distance performs better. For several mostly basic patterns the edit distance does still not discriminate between the languages. In some cases, this could be so because there virtually are no differences between the languages. Nevertheless, the edit distance discriminates in much more cases than the trivalent measure and clearly exceeds it.

WS-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 supports two service interaction patterns that are not supported by WS-BPEL and more than twice as many time patterns. Furthermore for almost all time and service interaction patterns, the solutions are less costly in WF. Altogether, from a pattern-support point of view it can be said that WF excels WS-BPEL.

For Sun BPEL, the analysis demonstrates that there is quite an amount of deficiencies in the language. Its support for control-flow, change, and service interaction patterns is rather limited. On the one hand, Sun BPEL does not implement several aspects of WS-BPEL 2.0 that are crucial for the support of these patterns. Such aspects are structures for graph-oriented modeling, parallel looping structures and a built-in synchronization primitive. On the other hand, the functioning of several important features, such as dynamic partner binding, that is supposed to work in Sun BPEL, could not be reproduced. At least for the time patterns, Sun

BPEL is able to increase the degree of support with the help of its additional time related functions. However, WF still supports twice as many time patterns as Sun BPEL. All in all, Sun BPEL obviously lags behind both WS-BPEL and WF.

6 Conclusion

The aim of this study is to propose an improvement of pattern-based analysis and test it by assessing the orchestration languages WS-BPEL and WF. Comparability and selectivity of pattern-based analysis are improved by a unified approach for judging the validity of a solution to a pattern in a language and using the edit distance to calculate the degree of support. The results demonstrate that this approach is able to overcome some of the problems of traditional analyses. For all pattern catalogs in focus, the approach provides a higher degree of selectivity of the results. As the edit distance is calculated in the same fashion for all catalogs, it is possible to directly compare the results between different catalogs. The measure also provides direct comparability between the different languages, because it is based on the same set of change operations for all languages. Future analyses can provide more meaningful and selective results by relying on the approach presented here.

The set of change operations also marks an area where the approach can be improved in future work. The set of operations presented here is fine-tuned for orchestration languages. It is very likely that other language types, such as choreography languages, require another set of change operations. The application of the approach for such other language types might bear interesting findings.

As for the assessment of the languages, the results show that WF excels both, the WS-BPEL standard and its implementation Sun BPEL concerning the degree of pattern support. In that sense, WF can be said to be more appropriate for the B2Bi domain. It would of course be beneficial to consider more orchestration languages and pattern catalogs in future work.

Apart from its main aim, the study provides several other contributions and areas of future work. One such contribution is the ranking of pattern-catalogs for B2Bi. In most studies, it is simply assumed that a pattern catalog is relevant. However there seem to be significant differences in the importance of the pattern catalogs. Unfortunately, this topic is seldom touched and more work is necessary on the subject of when a pattern or pattern catalog is really relevant to a domain or language. The ranking presented here is a first step towards deciding on the importance of pattern catalogs. When performing an analysis for a different domain, such as the medical industry, a ranking might look completely different.

Last but not least, the process models developed here can be a valuable help for people working with the two languages. They provide a comprehensive set of executable examples for a wide array of different aspects. As also stated by the reviewers of [24], the more complex process models, available at `http://www.uni-bamberg.de/en/pi/bereich/research/software-projects/pattern-based-analysis-of-orchestration-languages/` are of interest for a broader audience. For instance, they could be used as a resource in academic teaching.

References

[1] C. Alexander. A Pattern Language. Oxford University Press, August 1978. ISBN: 0195019199.

[2] A. P. Barros, G. Decker, M. Dumas, and F. Weber. Correlation Patterns in Service-Oriented Architectures. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 245–259, Braga, Portugal, March/April 2007.

[3] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service Interaction Patterns. In *3rd International Conference on Business Process Management*, pages 302–318, Nancy, France, September 2005.

[4] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. Technical report, Queensland University of Technology, Faculty of IT, Australia, 2005.

[5] B. Bukovics. *Pro WF: Windows Workflow in .NET 4*. Apress, June 2010. ISBN-13: 978-1-4302-2721-2.

[6] D. Chappell. The Workflow Way: Understanding Windows Workflow Foundation. Technical report, Microsoft Corporation, April 2009.

[7] G. Decker, O. Kopp, and A. P. Barros. An Introduction to Service Choreographies. *it - Information Technology, Oldenbourg Wissenschaftsverlag*, 50(2):122–127, 2008.

[8] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS)*, pages 296–303, Salt Lake City, Utah, USA, July 2007.

[9] G. Decker, O. Kopp, F. Leymann, and M. Weske. Interacting services: From specification to execution. *Data & Knowledge Engineering, Elsevier*, 68(10):946–972, 2009.

[10] G. Decker and J. Mendling. Process Instantiation. *Data and Knowledge Engineering, Elsevier*, 68:777–792, 2009.

[11] G. Decker, H. Overdick, and J. Zaha. On the Suitability of WS-CDL for Choreography Modeling. In *Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA)*, pages 21–33, Hamburg, Germany, October 2006.

[12] 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.

[13] R. M. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, and J. Mendling. Similarity of Business Process Models: Metrics and Evaluation. *Inf. Syst. (IS)*, 36(2):498–516, 2011.

[14] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[15] J. Dorn, C. Grün, H. Werthner, and M. Zapletal. A Survey of B2B Methodologies and Technologies: From Business Models towards Deployment Artifacts. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 143, Hawaii, USA, January 2007.

[16] T. Erl. *SOA: Entwurfsprinzipien für serviceorientierte Architektur*. Addison-Wesley, München, 2008. ISBN: 978-3-8273-2651-5.

[17] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Amsterdam, 1995. ISBN: 0201633612.

[18] J. Gray. Notes on Database Systems. Technical report, IBM, San Jose, California USA, February 1978. IBM Research Report RJ2188.

[19] C. Hentrich and U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–45, Irsee, Germany, July 2006.

[20] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, Amsterdam, 2004. ISBN: 0321200683.

[21] O. Kopp, D. Martin, D. Wutke, and F. Leymann. The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. *Enterprise Modelling and Information Systems, Gesellschaft für Informatik e.V. (GI)*, 4(1):3–13, 2009.

[22] A. Lanz, B. Weber, and M. Reichert. Time Patterns in Process-aware Information Systems: A Pattern-based Analysis - Revised version. Technical report, University of Ulm, Germany, 2009.

[23] A. Lanz, B. Weber, and M. Reichert. Workflow Time Patterns for Process-Aware Information Systems. In *Enterprise, Business-Process, and Information Systems Modelling: 11th International Workshop BPMDS and 15th International Conference EMMSAD in conjunction with CAiSE*, pages 94–107, Hammamet, Tunisia, June 2010.

[24] J. Lenhard, A. Schönberger, and G. Wirtz. Streamlining Pattern Support Assessment for Service Composition Languages. In *Proceedings of the 3rd Central-European Workshop on Services and their Composition, ZEUS 2011, Karlsruhe, Germany, 2011*, volume 705 of *CEUR Workshop Proceedings*, pages 112–119. CEUR-WS.org, February 2011.

[25] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[26] Microsoft. Overview of the .NET Framework. `http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx`, 2010.

[27] Microsoft. Windows Workflow Foundation. `http://msdn.microsoft.com/en-us/library/dd489441.aspx`, August 2010.

[28] N. Mulyar, W. M. P. van der Aalst, and L. Aldred. The Conceptualization of a Configurable Multi-party Multi-message Request-Reply Conversation. In *OTM Conferences (1)*, pages 735–753, Vilamoura, Portugal, November 2007.

[29] N. Mulyar, W. M. P. van der Aalst, L. Aldred, and N. Russell. Service Interaction Patterns: A Configurable Framework. In *BPM Center Report BPM-07-07*, 2007.

[30] A. Nysetvold and J. Krogstie. Assessing Business Processing Modeling Languages Using a Generic Quality Framework. In *Proceedings of the Workshop on Evaluating Modeling Methods for Systems Analysis and Design (EMMSAD'05), held in conjunctiun with the 17th Conference on Advanced Information Systems 2005 (CAiSE 2005)*, pages 545–556, Porto, Portugal, June 2005.

[31] OASIS. *Web Services Business Process Execution Language*, April 2007. v2.0.

[32] M. P. Papazoglou and D. Georgakopoulos. Service-oriented Computing. *Communications of the ACM*, 46(10):24–28, October 2003.

[33] C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, October 2003.

[34] N. Russell. *Foundations of Process-Aware Information Systems*. PhD thesis, Queensland University of Technology, June 2007.

[35] N. Russell, A. H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE05), volume 3520 of Lecture Notes in Computer Science*, pages 216–232, Porto, Portugal, June 2005. Springer.

[36] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns. Technical report, Queensland University of Technology, Brisbane, Australia, 2004.

[37] 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 *Proceedings of the 24th International Conference on Conceptual Modeling (ER2005)*, pages 353–368, Klagenfurt, Austria, October 2005. Springer.

[38] 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 Group, Queensland University of Technology; Department of Technology Management, Eindhoven University of Technology, 2006.

[39] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Workflow Exception Patterns. In *Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE06)*, pages 288–302, Luxembourg, Luxembourg, June 2006.

[40] A. Schönberger, C. Wilms, and G. Wirtz. A Requirements Analysis of Business-to-Business Integration. Technical Report 83, Otto-Friedrich-Universität Bamberg, Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik, December 2009. ISSN 0937-3349.

[41] A. Schönberger and G. Wirtz. Taxonomy on Consistency Requirements in the Business Process Integration Context. In *20th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 593–598, Redwood City, San Francisco Bay, USA, July 2008.

[42] L. H. Thom, M. Reichert, and C. Iochpe. Activity Patterns in Process-aware Information Systems: Basic Concepts and Empirical Evidence. *International Journal of Business Process Integration and Management (IJBPIM)*, 4(2):93–110, 2009.

[43] 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, July 2003.

[44] A. van Dijk. Contracting Workflows and Protocol Patterns. In *International Conference on Business Process Management*, pages 152–167, Eindhoven, The Netherlands, June 2003.

[45] W3C. *XML Path Language (XPath) Version 1.0*, November 1999. v1.0.

[46] W3C. *Web Services Description Language (WSDL) 1.1*, March 2001.

[47] W3C. *Web Services Architecture*, February 2004.

[48] W3C. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, 2007.

[49] W3C. *XML Path Language (XPath) 2.0 (Second Edition)*, December 2010.

[50] B. Weber, S. Rinderle, and M. Reichert. Change Support in Process-Aware Information Systems - A Pattern-Based Analysis. Technical report, University of Twente, The Netherlands, 2007.

[51] B. Weber, S. Rinderle, 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.

[52] 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 *22nd International Conference on Conceptual Modeling*, pages 200–215, Chicago, Illinois, USA, October 2003.

[53] P. Wohed, W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, and N. Russell. On the Suitability of BPMN for Business Process Modelling. In *Business Process Management*, pages 161–176, Vienna, Austria, September 2006.

[54] A. Wombacher and C. Li. Alternative Approaches for Workflow Similarity. In *Seventh International Conference on Services Computing*, pages 337–345, Miami, Florida, USA, July 2010.

[55] M. Zapletal, W. M. P. van der Aalst, N. Russell, P. Liegl, and W. H. An Analysis of Windows Workflow's Control-Flow Expressiveness. In *Seventh IEEE European Conference on Web Services ECOWS*, pages 200–209, Eindhoven, The Netherlands, November 2009.

[56] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, and F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *International Journal of Web and Grid Services*, 2(1):68 – 91, 2006.

A Overview of Support Measures

Pattern	WF 3.5 taken from [55]	WF 4	WS-BPEL 1.1 taken from [38]	WS-BPEL 2.0	Sun BPEL
Control-flow Patterns					
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	+	- (-)	+/-	12 (+/-)	- (-)
WCP-18. Milestone	+	11 (+/-)	-	11 (+/-)	11 (+/-)
WCP-39. Critical Section	+	9 (+/-)	+	11 (+/-)	40 (-)
WCP-40. Interleaved Routing	+	9 (+/-)	+	11 (+/-)	40 (-)
Cancellation Patterns					
WCP-19. Cancel Activity	+	7 (+/-)	+	6 (+/-)	6 (+/-)
WCP-20. Cancel Case	+	2 (+)	+	1 (+)	1 (+)
WCP-25. Cancel Region	+	7 (+/-)	+/-	6 (+/-)	6 (+/-)
WCP-26. Cancel MI Activity	+	12 (+/-)	-	11 (+/-)	51 (-)
WCP-27. Complete MI Activity	-	10 (+)	-	8 (+)	- (-)
Iteration Patterns					
WCP-10. Arbitrary Cycles	+	13 (+/-)	-	- (-)	- (-)
WCP-21. Structured Loop	+	5 (+)	+	5 (+)	5 (+)
WCP-22. Recursion	-	- (-)	-	- (-)	- (-)
Termination Patterns					
WCP-11. Implicit Termination	+	0 (+)	+	0 (+)	0 (+)
WCP-43. Explicit Termination	+	7 (+/-)	-	4 (+)	4 (+)
Trigger Patterns					
WCP-23. Transient Trigger	+	- (-)	-	- (-)	- (-)
WCP-24. Persistent Trigger	+	0 (+)	+	0 (+)	0 (+)
Change Patterns					
Patterns for Predefined Changes					
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 (+/-)		11 (+/-)	40 (-)
PP-4. Multiple Instance Activity		6 (+)		7 (+)	- (-)

Service Interaction Patterns					
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		- (-)		- (-)	- (-)
Time Patterns					
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 (-)

B Manual for WF Processes

All artifacts for the analysis of WF are contained in the corresponding archive available at `http://www.uni-bamberg.de/en/pi/bereich/research/software-projects/pattern-based-analysis-of-orchestration-languages/`.

B.1 Environment Installation

The execution and development environment for WF is the .NET framework in revision 4 and the Visual Studio 2010 Ultimate IDE. .NET 4 is freely available and can be downloaded at `http://msdn.microsoft.com/en-us/netframework/aa569263`. Visual Studio on the other hand is not for free. The version used in the study was obtained through a cooperation of the University of Bamberg and the MSDN Academic Alliance[33]. More information about Visual Studio 2010 Ultimate is available at `http://www.microsoft.com/visualstudio/en-us/products/2010-editions/ultimate`. For setting up the environment, please follow the installation instructions of the two products.

[33]For information about MSDNAA, please refer to `http://msdn.microsoft.com/en-us/academic/dd547439`.

B.2 Folder and Project Structure

The process files are contained in a Visual Studio solution. Such a solution bundles multiple projects. There is one such project for the process files for each of the four pattern catalogs and an additional project for classes and activities needed by the pattern projects. The name of the solution is “Pattern Analysis Service”. It is contained in a folder of the same name. It in turn contains five folders, `ChangePatterns`, `ControlFlowPatterns`, `ServiceInteractionPatterns`, `TimePatterns` and `Custom Activities`, as well as the solution file `Pattern Analysis Service.sln`. As the name suggests, four of the folders contain the project for the process files of one of the pattern catalogs. The fifth project, `Custom Activities`, contains XAML and C# files relevant to the other projects. All projects are C# projects. There is no XAML project in Visual Studio, so workflow services have to be developed either as part of a C# or a Visual Basic project. Each pattern project contains the project file, such as `ControlFlowPatterns.csproj`, and a `Web.config` file. The `Web.config` file is an XML file that contains workflow services related configurations which are default in all cases. Each of the C# projects again contains a number of sub-folders. There is one sub-folder for each pattern category. For example, the project for the control-flow patterns contains sub-folders for basic patterns, cancellation patterns, iteration patterns and so on. The process files for the patterns of a category are contained in the according folders. If a pattern is implemented by a single process, the respective process file is directly contained in the category folder. In some cases, as for the service interaction patterns, a pattern is implemented by more than one process files. In this case, there is one more sub-folder for each pattern with the name of the pattern in the category folder. This folder finally contains multiple process files that implement a single pattern. All process files are `.xamlx` files. The `Custom Activities` project is referenced by all other projects in their build path. The following files are contained in this project:

WorkflowStub.xamlx: An example implementation of the process stub as described in section 4.1.

LogCodeActivity.cs: A `CodeActivity` implemented in C# that is used in all workflow services for providing output of the process execution.

LogWriter.cs: A C# class used by `LogCodeActivity` for writing messages to the Windows Event Log.

The structure discussed in this paragraph is outlined in listing 46.

Listing 46: Project Structure for WF Processes

```
Pattern Analysis
          -| Custom Activities
                 LogCodeActivity.cs
                 LogWriter.cs
                 WorkflowStub.xamlx
          -| ChangePatterns
          -| ControlFlowPatterns
          -| ServiceInteractionPatterns
                 -| MultiTransmission
                        -| AtomicMulticastNotification
                        -| ContingentRequests
                        -| MultiResponses
                               MultiResponsesInitiator.xamlx
                               MultiResponsesResponder.xamlx
                 -| Routing
                 -| SingleTransmissionBilateralInteraction
```

```
                -| SingleTransmissionMultilateralInteraction
        -| TimePatterns
                -| DurationsAndTimeLags
                -| RecurrentProcessElements
                -| RestrictionsofProcessExecutionPoints
                -| Variability
                        TimeDependentVariability.xamlx
                TimePatterns.csproj
                Web.config
        Pattern Analysis Service.sln
```

B.3 Deploying and Executing Workflow Services

There are several ways in which workflow services can be deployed and executed. For the processes at hand, the deployment and execution process is very simple.

Loading the solution: First, it is necessary to load the solution containing the process files into Visual Studio. Given the environment is properly installed, this can be simply be done by double-clicking the solution file. **Important notice:** Each workflow service will require administrator privileges on its first execution (this point is made clear in the following section). So, it might be necessary to start Visual Studio with administrator privileges. After its first execution, the workflow service may also be executed with normal user privileges. When opening the solution for the first time, the workflow services will display errors in their definition. The `LogCodeActivity` is not yet available to the workflow services, because the solution has not been built yet. Building the solution also eliminates the error messages.

Deploying workflow services: All workflow services of the solution are deployed at once. This can be done by pressing the F5 key while Visual Studio is open with the solution loaded. This action compiles all files that have changed and redeploys changed files. On first deployment, one instance of the ASP.NET Development Server is started for each of the pattern projects. So, there is one server instance running that hosts all workflow services of a pattern catalog. Once all server instances have started and the services are deployed, pressing F5 simply recompiles and redeploys changed files. Each development server deploys the contents of a project directly for a certain port of localhost. Except for the service interaction patterns, this port is chosen dynamically by Visual Studio. The service interaction patterns on the other hand, will always be deployed on port 10000. If this port is already in use, deployment will fail.

Executing a workflow service: Once deployed, workflow services are accessible as Web services and therefore any Web service client software can be used to trigger their execution. The directory structure of deployed artifacts resembles the project structure. So, the workflow services will be available under a URL like `http://localhost:port/Basic/ExclusiveChoice.xamlx?wsdl`. It is most convenient to use the WCF Test Client for triggering the execution of a workflow service. A straight-forward way to fire up this Test Client is by selecting a `.xamlx` file in Visual Studio and pressing F5. This requires that also the project that contains the `.xamlx` file is marked as start project. This can be done by right-clicking the project in Visual Studio and selecting the option “mark as start project”. The use of the Test Client is fairly simply. It is sufficient to set optional input data and press “invoke” for one of the Web service operations. The operation that

starts the execution of the processes for a pattern is called `StartProcess` in almost all cases. To be able to modify project files, it is necessary to close the client. Some workflow services behave differently, depending on their input. For example, the number of activity instances to be executed is determined by the input. To verify that a process works as intended, it is therefore sometimes necessary to provide different input parameters. Most solutions for the service interaction patterns use input parameters as correlation identifiers.

B.4 Logging in Workflows

All workflow services use `LogCodeActivities` which write XML messages to the Windows Event Log[34]. The Windows Event Log is a logging database that is available in Windows operating systems. The use of such a database is necessary, because some patterns involve parallel or distributed execution of multiple `LogCodeActivities`. Applications may register log files and sources for the Windows Event Log. The structure of the logging activity used is fairly simple. The activity takes three arguments, a `Message`, a `LogName` being the name of the pattern for which the activity is executed, and a `MessageNumber`. The `MessageNumber` has a special meaning. All log messages of a workflow are ordered as defined by their number and should appear accordingly in the Windows Event Log. This means that it should never be the case that a message with number 2 appears in the Event Log after a message with number 3. Each message number corresponds to a phase in the execution of a process. A phase may comprise multiple log messages which have identical `MessageNumbers`. This is the case, because a phase may involve multiple parallel events, such as the reception of messages at two different process instances.

On execution, the `LogCodeActivity` creates an object of type `LogWriter` which provides a `WriteLine` method to write logging data to the Windows Event Log. This method is illustrated in listing 47.

Listing 47: WriteLine Method of the LogWriter Class

```
1 public void WriteLine(String Message,Int32 MessageNumber)
2         {
3             EventLog myEventLog = new EventLog();
4             myEventLog.Log = "PatternAnalysis";
5             myEventLog.Source = LogName;
6             if (!EventLog.SourceExists(myEventLog.Source))
7             {
8                 EventLog.CreateEventSource(myEventLog.Source,myEventLog.Log);
9             }
10
11            myEventLog.WriteEntry(Message,EventLogEntryType.Information,MessageNumber);
12        }
```

Lines 3 to 4 establish access to the log file. Line 5 sets the source of the log message to the name of the pattern which executes the activity. Lines 6 to 9 create the log file and event source, if they do not yet exist. Line 8 can only be executed with administrator privileges

[34] The documentation of the Windows Event Log is available at `http://msdn.microsoft.com/en-us/library/aa385780%28VS.85%29.aspx?ppud=4`.

which is why a workflow service for a certain pattern must be in possession of these privileges on its first execution. Without these privileges, the workflow exits with a `SecurityException`. If the `EventSource` already exists, line 8 is not executed and administrator privileges are not required. Finally, Line 11 writes the message to the Event Log.

B.5 Viewing Workflow Output

As demonstrated in the previous section, all processes write to a log file in the Windows Event Log named `PatternAnalysis`. The Windows Event Log can be accessed via the *Windows Event Viewer*. This viewer is to be found under `Control Panel / System and Maintenance / Administrative Tools` or can be opened by executing the command `eventvwr` on the command line. The log `PatternAnalysis` is to be found in the folder `Application and Services Logs`. All log entries have the log level "Information" and consist of a time stamp, the name of the log source and the message number. The message text is displayed when selecting a specific message. Figure 7 depicts an exemplary view of the `PatternAnalysis` log in the Windows Event Viewer. Using the Event Viewer, it is possible to check that the processes really execute as intended and conform to the execution trace of a pattern.

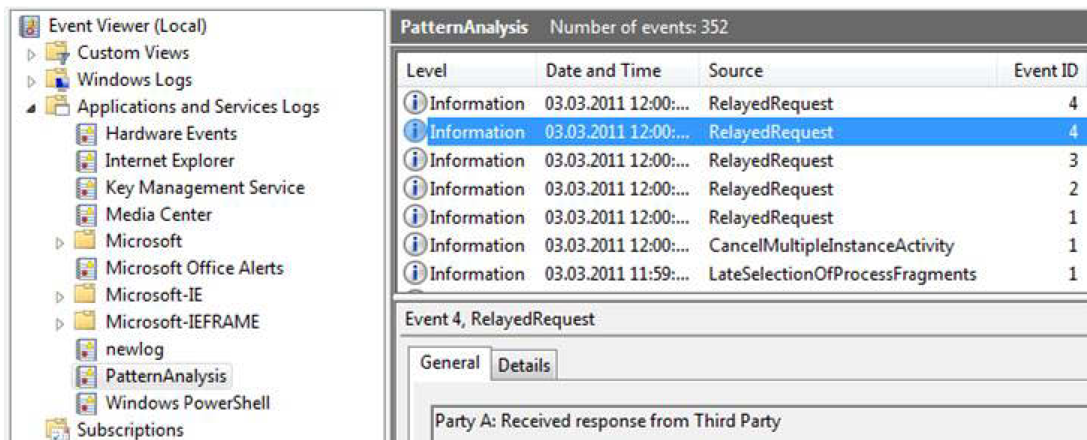


Figure 7: Analysis Log in the Event Viewer

C Manual for WS-BPEL Processes

All artifacts for the WS-BPEL and Sun BPEL processes can be found in the corresponding archive available at `http://www.uni-bamberg.de/en/pi/bereich/research/software-projects/pattern-based-analysis-of-orchestration-languages/`.

C.1 Environment Installation

Hopefully, the installer for the GlassFish ESB can be downloaded from the project page. Because of the relocation of resources from Sun Microsystems to Oracle, the installer became unavailable while performing the analysis. Therefore, no link to an installer can be provided here.

C.2 Folder and Project Structure

The archive contains three Netbeans projects. These are two BPEL projects named `Patterns` and `PatternTestBpelProject` and a composite application project named `PatternTestCompositeApp`.

Patterns: This project contains the process files developed during the analysis in its `src` folder. There is one sub-folder for each of the pattern catalogs. These sub-folders are structured similarly to the different C# projects in WF. The `src` folder also contains necessary WSDL and XSD files as well as the process stub used in the processes.

PatternTestBpelProject: This project is used as a container for WS-BPEL processes that are to be deployed. Its structure is similar to the `Patterns` project. It consists of several layers of empty folders to which process files can be copied.

PatternTestCompositeApp: This project can be deployed in the GlassFish ESB and references PatternTestBpelProject.

SoapUI: The folder contains an XML file which stores the configuration of a soapUI project that can be used as a client for triggering the execution of the pattern processes.

The structure discussed in this section is outlined in listing 48.

Listing 48: Project Structure for BPEL Processes

```
Patterns
           -| nbproject
           -| src
               -| Change
               -| ControlFlow
               -| ServiceInteraction
               -| Time
                    -| DurationsAndTimeLags
                    -| RecurrentProcessElements
                    -| RestrictionsOfProcessExecutionPoints
                    -| Variability
                    TimePattern.wsdl
               BpelStub.bpel
               Logging.xsd
               Partner.wsdl
```

```
                PartnerMessage.xsd
                Pattern.wsdl
                Properties.wsdl
            build.xml
            catalog.xml
PatternTestBpelProject
    -| nbproject
            -| src
                -| first
                     -| second
                          -| third
                BpelStub.bpel
                Logging.xsd
                Partner.wsdl
                PartnerMessage.xsd
                Pattern.wsdl
                Properties.wsdl
            build.xml
            catalog.xml
PatternTestCompositeApp
SoapUI
```

C.3 Related WSDL and XSD Files

Nearly all WS-BPEL processes share the same WSDL and XSD files to facilitate their maintenance and use. This is also the reason why they cannot all be deployed at the same time and additional projects are necessary. If every WS-BPEL process had an own WSDL interface, development and execution would be rather complex. Having most processes reference the same generic interfaces eases these problems. All related files are contained in the `src` folder of both BPEL Projects.

Logging.xsd: This XSD file defines the `complexType` used as type of log message variables in several processes. This `complexType` consists of three elements. A process identifier of type `int`, a message of type `string`, and the name of a pattern of type `string`.

Properties.wsdl: This WSDL file is very simple. It consists of a single `property` of type `int` which is used as common correlation identifier by the other WSDL files.

Pattern.wsdl: This is the most important WSDL file for the processes. It describes the `myRole` interface of almost all processes and is used as starting point for triggering the execution of the processes for a pattern. An exception to this are several service interaction patterns and all time patterns. The time patterns use an alternative generic WSDL interface that is to be found in the folder of the pattern catalog. This interface is needed, because time patterns require more specific messages (involving dates and time stamps) than other patterns. The `portType` of `Pattern.wsdl` contains two asynchronous and one synchronous operation. The messages transmitted by these operations have a single element of type `int`. The file also defines a `binding` and a `service` with a `port`. Also the `Properties.wsdl` file is imported and `propertyAliases` for the correlation identifier are defined.

Partner.wsdl and **PartnerMessage.xsd**: Analogous to the `Pattern.wsdl`, there is a generic interface for additional partners, in case more than one process is involved in the implementation of a pattern. Such a process uses this WSDL file as `myRole` interface. The `PartnerMessage` defines the messages that are used when communicating with a partner

process. The structure of this message in principle resembles the structure of a log message as defined in `Logging.xsd` and is therefore sometimes also used for logging purposes.

C.4 Deployment Procedure

The deployment process for WS-BPEL processes is more complex than for WF, because it involves the copying of process files. Here, also the two other projects "PatternTestBpelProject" and "PatternTestCompositeApp" come into play. Deployment should be performed using Netbeans. The documentation of the OpenESB located at `http://wiki.open-esb.java.net/Wiki.jsp?page=BPELDesignerAndServiceEngineOverview` can be of help.

Open projects: The first step is to open the relevant projects in Netbeans. This can be done by selecting the "Open Project" menu item in the "File" menu and navigating to the path where the projects are located. Once opened, the projects should reappear in Netbeans each time it restarts.

Start application server: Prior to deploying processes, it is necessary to start the GlassFish Application Server. This can be done with a start script that can be found in the installation folder of the application server, or in Netbeans. The GlassFish is to be found in the "Services" Tab under the menu "Servers"‘and can be started by right-clicking it and selecting the "Start" menu item[35]. Netbeans provides an output window for the server log (which opens on starting the server). The server log can also be found in the server domain[36].

Copy process files: To deploy a process, it is necessary to copy the relevant files into "PatternTestBpelProject". Necessary files can be a single WS-BPEL process file or multiple files and additional WSDL and XSD files, depending on whether a pattern is implemented by one or multiple processes. Normally, the naming and folder structure for a pattern in the main project provide this information. The "PatternTestBpelProject" provides a hierarchy of three layers of folders. Because the `import` definitions of the WS-BPEL processes depend on it, it is necessary to copy a process file to exactly the same level in the file system hierarchy.

Build and deploy composite application: The next step is to clean and build the composite application project "PatternTestCompositeApp". This project references "PatternTestBpelProject". In the GlassFishESB, WS-BPEL processes must be deployed as a part of a composite application. Cleaning and building can be triggered via the context menu of the project. If no errors occur while building (for example because the WS-BPEL process is not placed in the right directory level), the project can be deployed. This can also be done via the context menu of the composite application project.

[35]For some reason, on first start, it is necessary to right-click the server twice before the "Start" menu item becomes available.

[36]This log is located at `INSTALL-DIR/GlassFishESBv22/glassfish/domains/domain1/logs/server.log`

C.5 Executing Processes

Once deployed, processes are accessible as Web services and can therefore be started using any Web service client software. In the analysis, soapUI[37] was used. The folder SoapUI contains the configuration file for a soapUI project. There, the relevant WSDL files have been important and default input parameters for all operations are configured. It might be necessary to update these files before sending requests, to adjust the configuration to the local setting. The update function is available in the context menu of the two files. Of course each process needs to be started with the proper WSDL operation. Which one this is can be read from the process file. In almost all cases, this is the asynchronous operation `startProcessInstanceAsync`. Just as for WF, some processes behave differently, depending on the input delivered to the process.

C.6 Viewing Process Output

As discussed, WS-BPEL processes provide output using trace extensions, mostly in `assign` activities. The according log messages are written to the server log and can therefore be viewed in the corresponding output window in Netbeans or directly in the log file. The structure of the log messages resembles the structure of the variable that is used. An example output is demonstrated in listing 49.

Listing 49: Logging Output for WS-BPEL Processes

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <LogMessage xmlns="http://lspi.wiai.uniba.de/bpel/Logging">
3     <processId>3</processId>
4     <message>Executing Branch1</message>
5     <pattern>ParallelSplit</pattern>
6 </LogMessage>
7
8 <?xml version="1.0" encoding="UTF-8"?>
9 <LogMessage xmlns="http://lspi.wiai.uniba.de/bpel/Logging">
10     <processId>3</processId>
11     <message>Executing Branch2</message>
12     <pattern>ParallelSplit</pattern>
13 </LogMessage>
```

[37] SoapUi is a free Web Services testing software. It is available at `http://www.soapui.org/`.

D List of previous University of Bamberg reports

Bamberger Beiträge zur Wirtschaftsinformatik

Nr. 1 (1989)	Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)
Nr. 2 (1990)	Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG
Nr. 3 (1990)	Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen
Nr. 4 (1990)	Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)
Nr. 5 (1990)	Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)
Nr. 6 (1991)	Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten
Nr. 7 (1991)	Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender
Nr. 8 (1991)	Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung
Nr. 9 (1992)	Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell
Nr. 10 (1992)	Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)
Nr. 11 (1992)	Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells
Nr. 12 (1992)	Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt
Nr. 13 (1992)	Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell
Nr. 14 (1992)	Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen
Nr. 15 (1992)	Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch
Nr. 16 (1992)	Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip
Nr. 17 (1993)	Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation

Nr. 18 (1993) Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells

Nr. 19 (1994) Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach

Nr. 20 (1994) Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994

Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994

Nr. 21 (1994) Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen

Nr. 22 (1994) Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems

Nr. 23 (1994) Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle

Nr. 24 (1994) Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen

Nr. 25 (1994) Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme

Nr. 26 (1995) Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes

Nr. 27 (1995) Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995

Nr. 28 (1995) Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach

Nr. 30 (1995) Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit

Nr. 31 (1995) Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse

Nr. 32 (1995) Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger

Nr. 33 (1995) Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?

Nr. 34 (1995) Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -

Nr. 35 (1995) Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme

Nr. 36 (1996) Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996)	Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996
Nr. 38 (1996)	Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996
Nr. 39 (1996)	Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze
Nr. 40 (1997)	Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997
Nr. 41 (1997)	Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997
Nr. 42 (1997)	Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997
Nr. 43 (1997):	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997
	Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998
Nr. 44 (1997)	Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung
Nr. 45 (1998)	Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998
Nr. 46 (1998)	Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997
Nr. 47 (1998)	Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998
Nr. 48 (1998)	Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering
Nr. 49 (1998)	Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems

Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460

Nr. 50 (1999)	Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)
Nr. 51 (1999)	Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)
Nr. 52 (1999)	Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule" im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999
Nr. 53 (1999)	Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999
Nr. 54 (1999)	Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.
Nr. 55 (2000)	Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur
Nr. 56 (2000)	Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000
Nr. 57 (2000)	Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.
Nr. 58 (2000)	Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.
Nr. 59 (2001)	Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001
Nr. 60 (2001)	Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001" im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Note: The title of our technical report series has been changed from *Bamberger Beiträge zur Wirtschaftsinformatik* to *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* starting with TR No. 61

Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik

Nr. 61 (2002)	Goré R., Mendler M., de Paiva V. (Hrsg.): Proceedings of the International Workshop on Intuitionistic Modal Logic and Applications (IMLA 2002), Copenhagen, July 2002.
Nr. 62 (2002)	Sinz E.J., Plaha M., Ulbrich-vom Ende A.: Datenschutz und Datensicherheit in einem landesweiten Data-Warehouse-System für das Hochschulwesen. Erscheint in: Beiträge zur Hochschulforschung, Heft 4-2002, Bayerisches Staatsinstitut für Hochschulforschung und Hochschulplanung, München 2002
Nr. 63 (2005)	Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions
Nr. 64 (2005)	Ferstl, O.K.: Lebenslanges Lernen und virtuelle Lehre: globale und lokale Verbesserungspotenziale. Erschienen in: Kerres, Michael; Keil-Slawik, Reinhard (Hrsg.); Hochschulen im digitalen Zeitalter: Innovationspotenziale und Strukturwandel, S. 247 – 263; Reihe education quality forum, herausgegeben durch das Centrum für eCompetence in Hochschulen NRW, Band 2, Münster/New York/München/Berlin: Waxmann 2005
Nr. 65 (2006)	Schönberger, Andreas: Modelling and Validating Business Collaborations: A Case Study on RosettaNet
Nr. 66 (2006)	Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz: Concealing Presence Information in Instant Messaging Systems, April 2006
Nr. 67 (2006)	Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz: Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems, April 2006
Nr. 68 (2006)	Michael Mendler, Thomas R. Shiple, Gérard Berry: Constructive Circuits and the Exactness of Ternary Simulation
Nr. 69 (2007)	Sebastian Hudert: A Proposal for a Web Services Agreement Negotiation Protocol Framework . February 2007
Nr. 70 (2007)	Thomas Meins: Integration eines allgemeinen Service-Centers für PC-und Medientechnik an der Universität Bamberg – Analyse und Realisierungs-Szenarien. Februar 2007
Nr. 71 (2007)	Andreas Grünert: Life-cycle assistance capabilities of cooperating Software Agents for Virtual Enterprises. März 2007
Nr. 72 (2007)	Michael Mendler, Gerald Lüttgen: Is Observational Congruence on μ-Expressions Axiomatisable in Equational Horn Logic?
Nr. 73 (2007)	Martin Schissler: out of print
Nr. 74 (2007)	Sven Kaffille, Karsten Loesing: Open chord version 1.0.4 User's Manual. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 74, Bamberg University, October 2007. ISSN 0937-3349.

Nr. 75 (2008) Karsten Loesing (Hrsg.): Extended Abstracts of the Second *Privacy Enhancing Technologies Convention* (PET-CON 2008.1). Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 75, Bamberg University, April 2008. ISSN 0937-3349.

Nr. 76 (2008) Gregor Scheithauer and Guido Wirtz: Applying Business Process Management Systems? A Case Study. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 76, Bamberg University, May 2008. ISSN 0937-3349.

Nr. 77 (2008) Michael Mendler, Stephan Scheele: Towards Constructive Description Logics for Abstraction and Refinement. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 77, Bamberg University, September 2008. ISSN 0937-3349.

Nr. 78 (2008) Gregor Scheithauer and Matthias Winkler: A Service Description Framework for Service Ecosystems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 78, Bamberg University, October 2008. ISSN 0937-3349.

Nr. 79 (2008) Christian Wilms: Improving the Tor Hidden Service Protocol Aiming at Better Performances. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 79, Bamberg University, November 2008. ISSN 0937-3349.

Nr. 80 (2009) Thomas Benker, Stefan Fritzemeier, Matthias Geiger, Simon Harrer, Tristan Kessner, Johannes Schwalb, Andreas Schönberger, Guido Wirtz: QoS Enabled B2B Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 80, Bamberg University, May 2009. ISSN 0937-3349.

Nr. 81 (2009) Ute Schmid, Emanuel Kitzelmann, Rinus Plasmeijer (Eds.): Proceedings of the ACM SIGPLAN Workshop on *Approaches and Applications of Inductive Programming* (AAIP'09), affiliated with ICFP 2009, Edinburgh, Scotland, September 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 81, Bamberg University, September 2009. ISSN 0937-3349.

Nr. 82 (2009) Ute Schmid, Marco Ragni, Markus Knauff (Eds.): Proceedings of the KI 2009 Workshop *Complex Cognition*, Paderborn, Germany, September 15, 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 82, Bamberg University, October 2009. ISSN 0937-3349.

Nr. 83 (2009) Andreas Schönberger, Christian Wilms and Guido Wirtz: A Requirements Analysis of Business-to-Business Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 83, Bamberg University, December 2009. ISSN 0937-3349.

Nr. 84 (2010) Werner Zirkel and Guido Wirtz: A Process for Identifying Predictive Correlation Patterns in Service Management Systems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 84, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 85 (2010) Jan Tobias Mühlberg und Gerald Lüttgen: Symbolic Object Code Analysis. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 85, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 86 (2010) Werner Zirkel and Guido Wirtz: Proaktives Problem Management durch Eventkorrelation – ein *Best Practice* Ansatz. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 86, Bamberg University, August 2010. ISSN 0937-3349.

Nr. 87 (2010) Johannes Schwalb, Andreas Schönberger: Analyzing the Interoperability of WS-Security and WS-ReliableMessaging Implementations. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 87, Bamberg University, September 2010. ISSN 0937-3349.

Nr. 88 (2011) Jörg Lenhard: A Pattern-based Analysis of WS-BPEL and Windows Workflow. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 88, Bamberg University, March 2011. ISSN 0937-3349.