# Bridging the Heterogeneity of Orchestrations – A Petri Net–based Integration of BPEL and Windows Workflow

Stefan Kolb, Jörg Lenhard, and Guido Wirtz
*Distributed Systems Group*
*University of Bamberg*
*Bamberg, Germany*
{*stefan.kolb, joerg.lenhard, guido.wirtz*}@*uni-bamberg.de*

***Abstract*—Service orchestrations are a powerful tool for implementing intra– and interorganizational business processes that base on services. Several heterogeneous orchestration languages can be found in contemporary IT landscapes. While the *Web Services Business Process Execution Language* (BPEL) is the de facto standard, others gain attention, including *Windows Workflow* (WF) in the .NET segment. When integrating orchestrations, incompatibilities between them can easily arise. In this paper, we investigate an automated Petri net–based integration between BPEL and WF to solve these issues with behavioral adapter services. We provide a mapping of WF to *Open Workflow Nets* (oWFNs) and implement it in a compiler. Thereby, we integrate our approach with existing approaches for BPEL and adapter synthesis and validate the integration with a standards–based case study using the two languages.**

***Keywords*-Orchestration, WF, BPEL, Adapter, Petri Nets, Service Integration**

## I. Introduction

In the last decade, information systems have become *process–aware*, with processes serving as a starting point for an implementation of intra– and interorganizational systems [1]. At the same time, there is an increasing acceptance of *Service–oriented Architectures* (SOAs) and Web Services as a suitable architectural paradigm and implementation technology for these systems [2]. Here, processes are implemented as composite services using orchestration languages. Within the scope of Web Services–based orchestration languages, BPEL [3] is the de facto standard, but other languages are gaining attention, especially WF [4] in the Windows segment.

To achieve a high degree of automation in a company's business processes as well as in B2Bi, such orchestrations need to be integrated. Basically, two integration strategies exist: top–down integration, where orchestrations are derived during the implementation phase, and bottom–up integration, where preexisting orchestrations need to communicate. Here, we focus on bottom–up integration. During this type of integration, incompatibilities such as unexpected blocking and deadlocks between the different orchestrations may occur.

Most process languages, including BPEL and WF, are informally defined and therefore are insufficient for formally tackling these issues. This can be solved by a transformation into an analyzable representation. Petri nets have proven to be applicable in the modeling and analysis of business processes [5], [6], [7]. However, current transformation approaches only cover BPEL [8], [9], [10] and omit several relevant languages, including WF. Adapter services that bridge the incompatibilites between existing orchestrations have been proposed for some time [11], [12], [13], [14], [15], [16]. In [11], oWFNs are used as basis for adapter synthesis. We extend the application of Petri net–based analysis methods and synthesis to WF by defining a transformation from WF to equivalent Petri net patterns, thereby integrating WF into the existing approaches. We have implemented this transformation in a compiler that is publicly available[1]. Moreover, we verify the practical applicability of our approach with an integration between BPEL and WF by means of oWFNs.

In Sec. II, we introduce underlying formalisms, including oWFNs and an algorithm for adapter synthesis. Moreover, we give an overview of Windows Workflow and associated concepts. Sec. III presents a subset of the derived Petri net patterns of our WF mapping. In Sec. IV, we describe a use case that integrates BPEL and WF processes based on existing tools and our WF compiler implementation. Finally, in Sec. V, we draw conclusions and discuss future work.

## II. Basics and Related Work

### A. Adapter Synthesis and oWFNs

Typical sources of incompatibilities that can occur when integrating existing orchestrations are [17]:

- names of the message types
- encoding of similar message types
- semantics of similar message types
- order in which messages are expected or transmitted

In such a scenario it is possible to solve incompatibilities by exchanging incompatible services with compatible ones, changing the implementation of services or introducing adapter services that solve the problems. In most of the situations only the latter can be applied because existing

[1]The project homepage is https://github.com/uniba-dsg/wf2owfn

implementations should not be changed. [18], [11] propose an approach for the synthesis of appropriate adapter services. The specification of an adapter includes the models of the services, and as behavioral property deadlock–freedom of the resulting composition. To be sure that the adapter only uses adequate message transformations and does not randomly delete or create messages, those can be specified through the *Specification of Elementary Adapter Activities* (SEA). Fig. 1 shows the proposed synthesis concept.

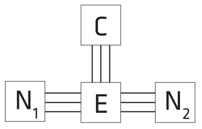

Figure 1. Conceptual adapter structure from [11]: The Services $N_1$ and $N_2$ denote two services that are integrated by an adapter that consists of an engine $E$ and a controller $C$.

To model services, we use oWFNs [19], [20] that generalize classical workflow nets [5]. Services are represented as distinct nets with an internal part and explicitly declared interface places to allow for asynchronous interaction between different nets.

*Definition 2.1:* oWFN [11]: An oWFN $N$ consists of

- two finite and disjoint sets $P$ (of places) and $T$ (of transitions),
- two disjoint subsets $P_i$ (of input places) and $P_o$ (of output places) of $P$,
- a flow relation $F$ such that $F \subseteq ((P \setminus P_o) \times T) \cup (T \times (P \setminus P_i))$,
- an initial marking $m_0$ such that $m_0.p = 0$ for all $p \in P_i \cup P_o$, and
- a set $\Omega$ of final markings such that no final marking enables any transition in $T$, and such that $m.p = 0$ for all $m \in \Omega$ and for all $p \in P_i \cup P_o$.

Several oWFN services can be composed by merging their equally named, interface–compatible places. The composition is called *controllable* if the resulting net is deadlock–free. The adapter between the services consists of a composition of the engine and a suitable controller. The engine in particular encodes the elementary activities, being the permissable message transformations. The controller determines the order in which the transformations can be applied. A controller can be synthesized as an arbitrary service that results in a controllable composition between itself, the services, and the engine. If the entire composition of the services and the adapter is deadlock–free, the adapter is correct by design. For details regarding the synthesis algorithm, we refer to [18], [11]. The tool MARLENE[2] implements this algorithm.

To apply the algorithm to orchestration languages, it is necessary to map their, mostly XML–based, representation to equivalent oWFNs. In case of BPEL there exists a feature–complete semantics [8] as well as corresponding tools for transforming BPEL to and from oWFNs[3]. We integrate WF into the toolchain by supplying a transformation to oWFNs. This transformation extends that of BPEL by a support for cyclic graph structures and state machines.

### B. Windows Workflow

WF is a technology that allows to implement long running processes as workflows within the .NET framework. It supports Web Services–based communication which makes it a natural candidate for building orchestrations. WF was first introduced with .NET Framework 3. Our research covers the latest available version 4.03. WF is tightly integrated into the .NET framework and the Visual Studio development environment featuring a workflow designer and a workflow engine. WF contains numerous built–in standard activities and the possiblity to create user–defined custom activities through code classes or the orchestration of existing activities. In contrast to most other workflow languages [21], WF provides several different modeling techniques to describe the control– and data–flow. Besides the commonly used block–structured style, termed sequential in WF, the language also supports a graph–based style (`Flowchart`) and finite state machines (`StateMachine`). Its control–flow expressiveness has been thoroughly examined [22], [23] and its applicability has been demonstrated in several settings, such as dynamic workflow adaption [24]. WF is mapped to the *Extensible Application Markup Language*[4] (XAML) and leverages a derivation of the vocabulary of the XAML language to describe control– and data–flow. Unlike in the case of BPEL, no semi–formal specification of WF's vocabulary is publicly available yet. Microsoft recently published the general specification of the XAML language and released several concrete vocabularies[5], which indicates that WF may also be disclosed later on. No public specification being available, we determined a semi–formal specification of the activities through unit tests and WF's API reference[6]. Based on this, we develop a mapping of the WF activities to equivalent oWFN patterns. Combined with the existing work for BPEL, this enables us to extend the approach for adapter synthesis to scenarios that involve the implementations of orchestrations in heterogeneous language settings.

## III. NET PATTERN AND TRANSFORMATION

The start of an integration of WF into current approaches for adapter synthesis is a translation of orchestration models in that language to an oWFN representation. Here, we adapt the hierarchical approach from [8] and use the same serialization formats which enables to rely on existing toolchains.

[2]Available at http://download.gna.org/service-tech/marlene/

[3]The tools are available at http://www.service-technology.org

[4]See http://www.microsoft.com/en-us/download/details.aspx?id=19600

[5]Examples are *Windows Presentation Foundation* (WPF) and Silverlight.

[6]Available at https://github.com/uniba-dsg/wf2owfn/downloads/

A WF process is built by orchestrating several WF activities. Similar to [8], we provide a *pattern*, an equivalent oWFN representation, for each WF activity. All patterns share a uniform interface (see Fig. 2). The translation process first compiles each activity into its oWFN representation and then joins all nets with the help of their interfaces.

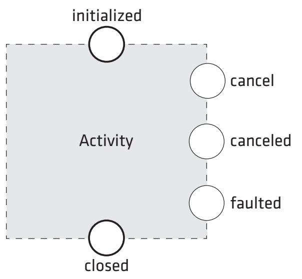

Figure 2. The interface places derived from the activity states [4, p.572]: *initialized*, *closed*, *cancel*, *canceled*, and *faulted*. The *initialized* place starts an activity and upon faultless completion of the activity, the *closed* place is marked. Both of these places are bold as they mark the successful completion of an activity. The places *cancel* and *canceled* model the premature termination of an activity. Faults are signaled by marking the *faulted* place.

Since our transformation approach is based on [8], it also shares the same constraints. Being low–level nets, oWFNs and our derived patterns abstract from data and time–related aspects. Data–dependent decisions, as used in looping activities, are therefore modeled nondeterministically, due to the indistinguishability of the tokens. This behavior only weakly preserves controllability of the initial process. If an oWFN is controllable then the original process is also controllable. The opposite does not hold in general. A process may be controllable because of the data aspects involved in the decisions, the transformed oWFN may be not, because aspects relevant to the control–flow were abstracted in the low–level representation. Moreover, currently only faultless behavior is modeled.

Our transformation focuses on WF's control–flow activities, except `ForEach<T>` and `ParallelForEach<T>`. These looping activities depend on a collection data structure. Their patterns are undecidable because of the possibly infinite data domain that determines the loop count [8]. All other activities can be covered with existing patterns or are out of the current scope (fault, compensation, and termination handling). We cannot present all derived patterns here due to space restrictions. Therefore, we limit the coverage to those activities that are relevant for the use case in Sec. IV and the modeling styles that are special for WF, `Flowchart` and `StateMachine`.

## A. Primitive Activities

Primitive activities perform atomic tasks such as sending or receiving a message. Focusing on normal control–flow, the primitive activities (`WriteLine`, `Delay`, `Assign`) can be reduced to the pattern given in Fig. 3(a). The

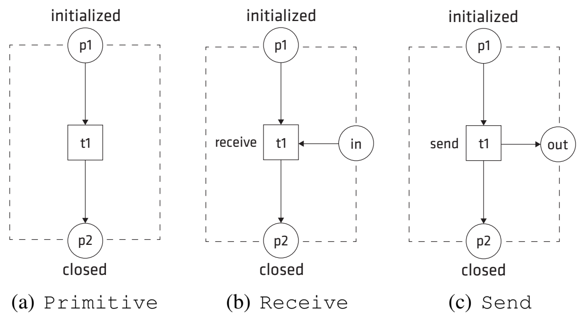

Figure 3. Primitives and Messaging Patterns

`Receive` activity models the reception of a single message and the `Send` activity the transmission, as demonstrated in Figs. 3(b) and 3(c). Moreover, there exist two activities that provide a combination of both – `ReceiveAndSendReply` and `SendAndReceiveReply`. These activities model a request–response pattern between two services. The patterns of both can be decomposed into a sequential combination of the single message patterns shown in Fig. 3.

## B. Sequential Style

Structured activities describe the control–flow of a process in a block–structured way. The simplest activity is the `Sequence` (see Fig. 4). Its purpose is to execute the contained activities one after another. In this pattern, the final place of an activity is also the initial place of the subsequent activity.

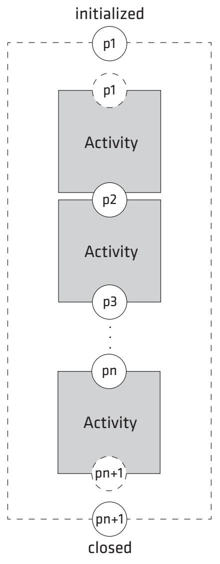

Figure 4. `Sequence` Pattern

The `If` activity is used to implement conditional branching in a process, based on a boolean condition. In the pattern (see Fig. 5) the decision of which path to take is modeled nondeterministically due to the indistinguishability of the tokens.

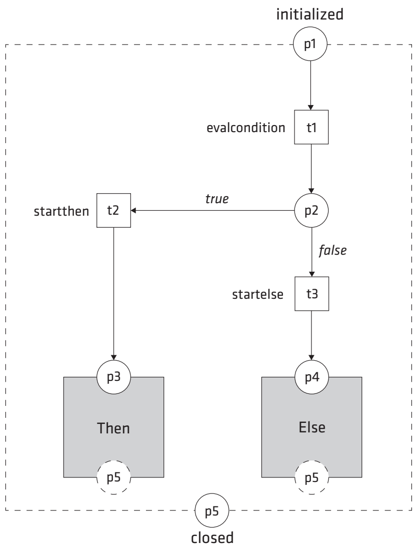

Figure 5. `If` Pattern

The `Pick` activity is designed to react to external events, such as incoming messages. It can include alternative branches (`PickBranches`) that include a trigger and an action activity. Upon completion of this trigger, all other branches are disabled and the corresponding action activity is executed. In contrast to BPEL, where only incoming messages or timing events are allowed as a trigger [3, pp.100-102], any activity can be used in WF. All triggers are scheduled for parallel execution. The trigger that completes its execution first determines the branch that is activated. This can be problematic if a trigger activity has side–effects. For this reason, Microsoft advises to only use a single atomic task as a trigger activity[7]. In order to avoid such problems due to partial execution of trigger activities, we reflect this situation in the pattern by modeling the pick activity as if only one trigger can actually fire. As a consequence, the `Pick` activity behaves in the same fashion as the `pick` in BPEL. Fig. 6 shows the Petri net pattern for the `Pick` activity. Both initial trigger places are merged with the initial place of the activity to only allow one trigger to fire.

## C. Flowchart Style

Using the `Flowchart` modeling style, WF supports graph–based control–flow definition. Here, control–flow is specified by directly linking activities. Since a `Flowchart` is a normal activity in WF, this enables the combination of the several modeling styles in a single orchestration. On the conceptual level, a `Flowchart` consists of several different `FlowNodes` connected with each other. A `FlowNode` is an abstract super class for the concrete manifestations `FlowStep`, `FlowDecision`, and `FlowSwitch<T>`. `FlowStep` acts as a container for any WF activity. It can be directly linked to another

[7]See http://msdn.microsoft.com/en-us/library/ee358746.aspx

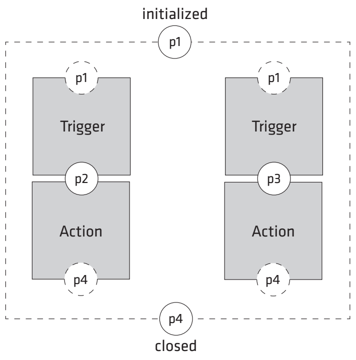

Figure 6. `Pick` Pattern with Two `PickBranches`

node in the graph. `FlowDecision` and `FlowSwitch<T>` model conditional branching in the context of `Flowchart` workflows. Fig. 7 shows an exemplary pattern containing these constructs. `FlowDecision` implements branching in a fashion that is equivalent to an `If` activity, whereas `FlowSwitch<T>` introduces several possible cases including an optional default case.

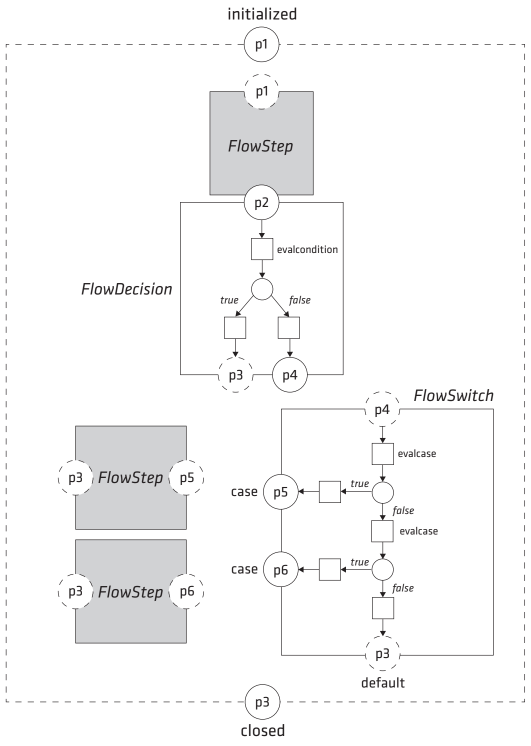

Figure 7. Exemplary `Flowchart` Pattern

## D. State Machine Style

Third comes WF's representation for state machines. In this style, states and event–based transitions among them are defined. Fig. 8 shows an exemplary pattern for a specific state machine. It consists of four states that are linked with each other. Links represent transitions among the states. The state machine can be refined to patterns for a particular state and for included transitions.

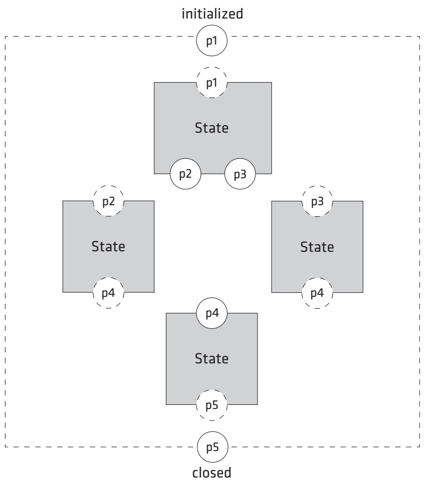

Figure 8. Exemplary Pattern of a `StateMachine`

Fig. 9 depicts the pattern of a state including two transitions. At first an entry activity is executed. After that, the state machine waits for an event to fire one of the transitions[8]. After a transition has completed, the state machine waits for the execution of an exit activity and traverses to the next state.

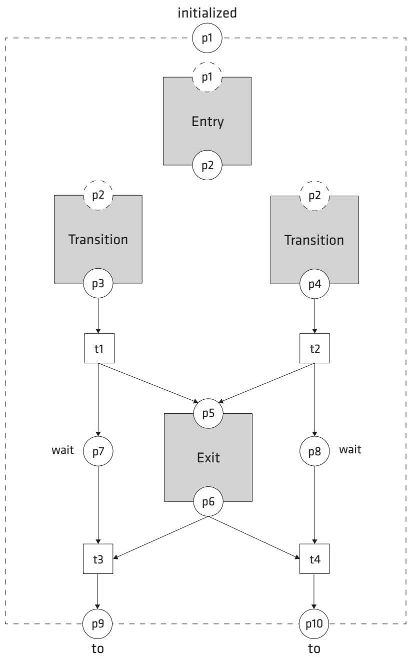

Figure 9. `State` Pattern with Two `Transitions`

A transition is by default assembled of a trigger activity, an expression that is evaluated before the transition fires, and an action activity which will be executed in case of success.

[8]Here, the same constraints regarding the trigger activities apply as stated for the `Pick` activity in Sec. III-B.

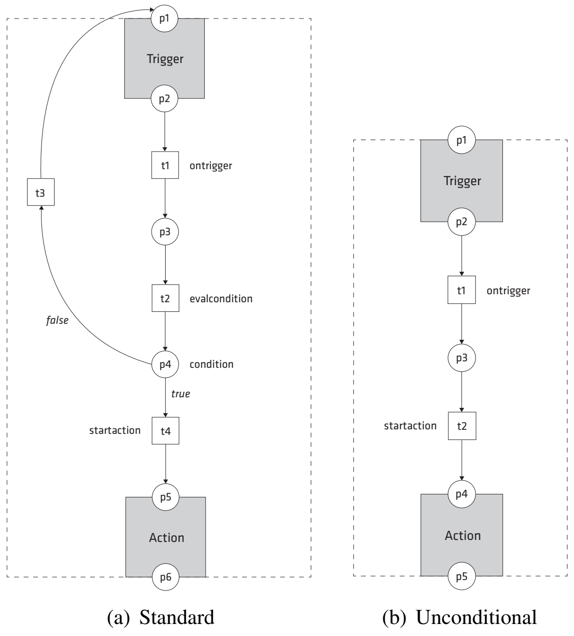

Figure 10. `Transition` Patterns

There do exist three variations of this default case: *unconditional trigger*, *shared trigger*, and *null trigger*. Fig. 10(a) shows the default behavior. Similar to the `Pick` activity a transition is fired by a trigger activity. Additionally, an expression is evaluated before the action is executed. If the expression does not evaluate to true, the flow of control is returned to its initial place. Several transitions may also share the same trigger activity, forming a shared trigger. Here, each transition is associated with an expression. These expressions are evaluated in order of their definition. Every transition has an action which is executed for the first transition for which the expression evaluates to true. If none of the expressions evaluate to true, the trigger behaves as in the default case. It is possible to omit the expression. This is an *unconditional trigger* (see Fig. 10(b)). After the completion of the trigger activity, the action is executed immediately. As a last variation, it is allowed to have at most one transition in a state that has no trigger and no expression. This *null trigger* immediately transfers the state to the next one after the entry activity of the state has completed.

## E. Prototype

We have implemented the patterns of the activities[9] above and several more in the compiler WF2OWFN[10]. This compiler demonstrates the feasibility of the patterns and can be integrated with other toolchains[11] to synthesize adapters for

[9]The implemented patterns already include further net optimizations. Moreover, for enhanced readability, the use case's nets were manually abbreviated in this paper.

[10]Available at https://github.com/uniba-dsg/wf2owfn

[11]See http://www.service-technology.org for an overview

composite services built in different languages. It is licensed under LGPL and is modularly structured in order to allow a simple extensibility with custom activities or the addition of further standard activities. A large set of test cases for all activities supported by the compiler is also provided and it has been tested with two process libraries[12].

## IV. USE CASE

To evaluate the toolchain with a realistic scenario, we have implemented the complete adapter synthesis using a process from the *Universal Business Language Specification 2.1* (UBL) [26] draft. In particular, we used the ordering process which models order processing between two parties (see Fig. 11). UBL is an open specification of business documents developed by OASIS. To overcome interoperability issues, UBL defines a generic extendable XML–format for business documents, and also defines processes in which these documents can be used. We use a slightly modified version of the ordering process depicted in Fig. 11.

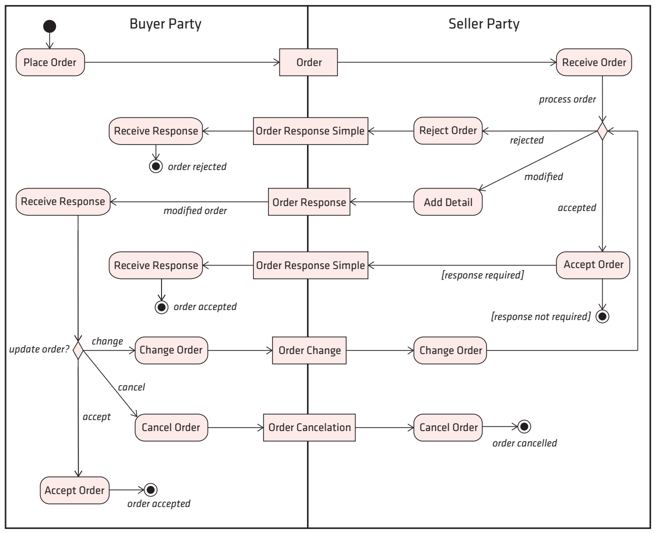

Figure 11. UBL Ordering Process [26].

In this process, two parties, a buyer and a seller, interact with each other to place an order. They do this by using several UBL–defined documents (in rectangulars). An order by the buyer may be accepted or rejected by the seller. Moreover, both parties may modify the order and add details to it, and the buyer may also cancel it. UBL provides normative semantics for the documents exchanged, but does not extend this requirement to the control–flow of the exemplary processes. We have implemented two concrete process definitions for the partners, one in BPEL and one in WF that are a priori incompatible and solve this issue later on with an adapter.

[12]Tested with Microsoft's WF 4 samples from http://www.microsoft.com/en-us/download/details.aspx?id=21459 and WF pattern implementations from [25].

To introduce incompatibility between the services, we removed the support for a buyer–initiated order modification in the seller process and implemented it in WF. Moreover, we added a confirmation message for a modified order from the seller in the buyer process. Otherwise, the buyer process was implemented in BPEL as defined. With the help of the tool BPEL2OWFN[13] we translated the BPEL process into an oWFN (see Fig. 12). Fig. 13 shows the corresponding oWFN of the seller, translated using our compiler and the patterns described in Sec. III.

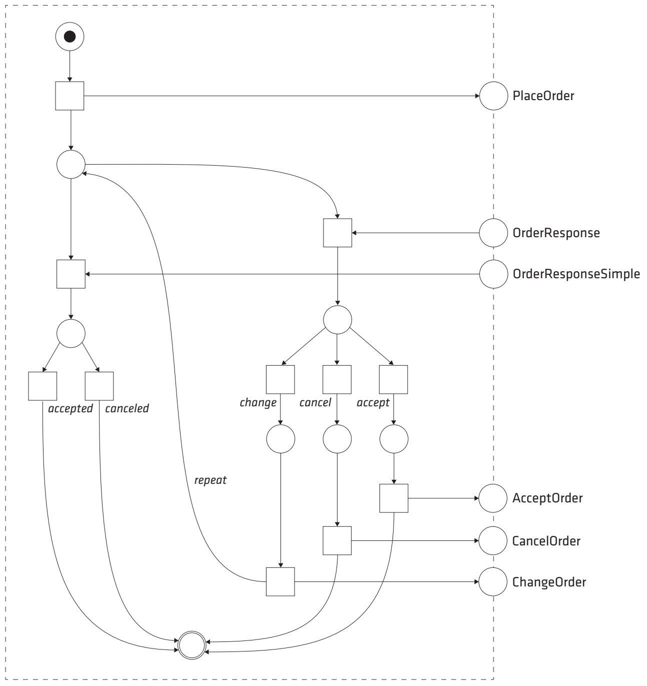

Figure 12. Buyer oWFN

As expected, the nets are not yet compatible. The buyer supports the possibility to modify an order of the seller, whereas the seller does not, resulting in a deadlock on the buyer side. All other operations are supported by both partners. To generate an adapter, the definition of all legitimate message transformations, the SEA, by the adapter is still required. These rules currently need to be specified manually [11]. To enable adapters to intercept all messages, the interfaces of the given services have to be disjoint [11]. Therefore, all input and output places are prefixed by their services (i.e. *b* for buyer and *s* for seller). As a consequence, even when using the same message types we need to supply virtual transformation rules. Rules one to five in Table I define the correct transfer of the messages from partner to partner. The lack of support for a modification of the seller's offer by the buyer must be solved with the adapter. One way is to allow the adapter to automatically cancel the buyer's request. As the seller on his part requires an

[13]Available at http://download.gna.org/service-tech/bpel2owfn/

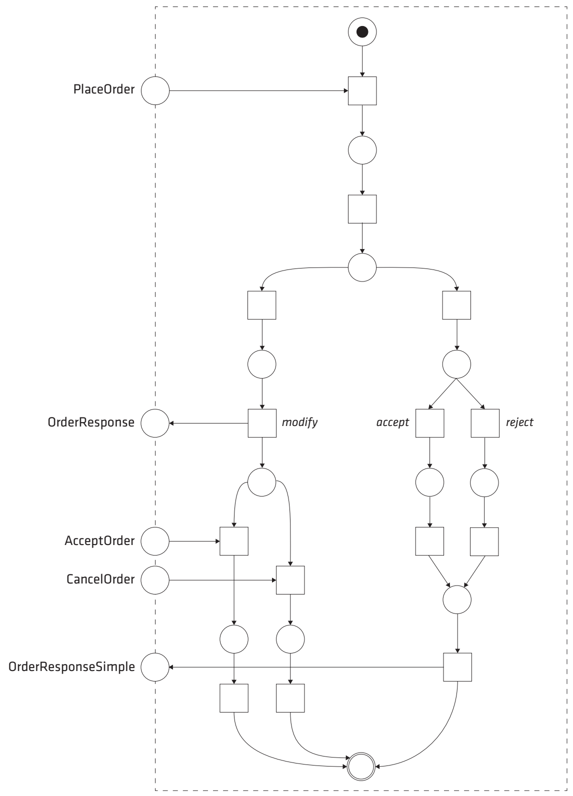

Figure 13. Seller oWFN

answer for his modified order, the adapter also needs to generate a cancelation for this request. This resolves the incompatibility, leading to a termination of both partners with an order cancelation. Rule six implements this by splitting an OrderChange into a cancelation message for both the buyer and the seller.

Table I
SEA ORDERING ADAPTER

| | | | |
|---|---|---|---|
| 1 | s.OrderResponseSimple | $\mapsto$ | b.OrderResponseSimple |
| 2 | s.OrderResponse | $\mapsto$ | b.OrderResponse |
| 3 | b.PlaceOrder | $\mapsto$ | s.PlaceOrder |
| 4 | b.AcceptOrder | $\mapsto$ | s.AcceptOrder |
| 5 | b.CancelOrder | $\mapsto$ | s.CancelOrder |
| 6 | b.ChangeOrder | $\mapsto$ | s.CancelOrder, b.OrderResponseSimple |

Using these rules, it is possible to create a correct adapter with the help of the tool MARLENE (see Sec. II). Fig. 14 shows the generated adapter net.

In order to get the composition running, we manually transformed the generated Petri net adapter into an executable WF process and tested the correctness of the interaction during operation. Moreover, we used the tool OWFN2BPEL[14] to automatically transform the net into an abstract BPEL process and implemented it.

[14]Available at http://download.gna.org/service-tech/owfn2bpel/

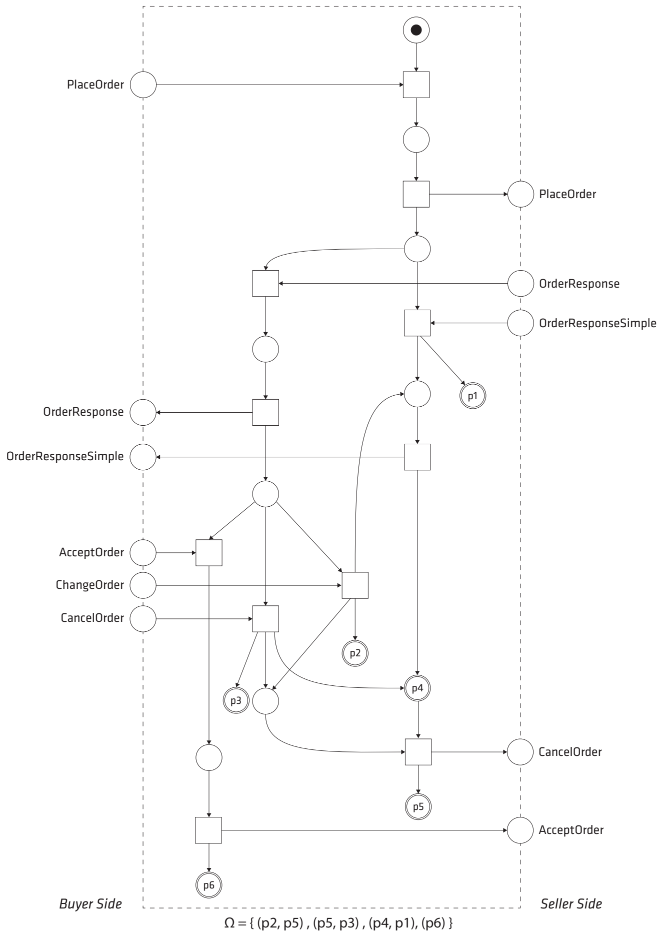

Figure 14. Adapter net that solves the issues between the buyer and seller nets.

Apart from several technical flaws in the compiler OWFN2BPEL, this integration proved to be feasible under the present restrictions. However, as the general approach does abstract from data and time, this can actually lead to false negative results as it only weakly preserves controllability of the initial process (see Sec. III). This particularly concerns data–dependent constructs like looping activities which may lead to uncontrollable nets due to the loss of potentially control–flow relevant data decisions. Therefore, we state that for true practical applicability the concept has to be broadened to include data and time.

## V. CONCLUSION AND FUTURE WORK

In this paper, we presented an automated Petri net–based integration between two process–based systems running BPEL and WF. We provided Petri net patterns for a large set of WF's standard activities, enabling it to work with current toolchains for adapter synthesis and implemented the translation of WF to oWFN patterns with the compiler WF2OWFN. As an evaluation of the applicability of the toolchain, we used a realistic ordering process from the UBL [26]. In the future we intend to add missing activities to provide a feature–complete semantics for WF. Moreover, we want to extend all activities with fault and cancelation logic to depict the complete control–flow. Finally, the semantics should be extended to capture data and time aspects to fully preserve controllability between the executable and the analyzable representations.

## REFERENCES

[1] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, "Business process management: A survey," in *Business Process Management*, ser. LNCS. Springer, 2003, vol. 2678, pp. 1–12.

[2] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented Computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, October 2003.

[3] OASIS, "Web Services Business Process Execution Language Version 2.0," OASIS, April 2007.

[4] B. Bukovics, *Pro WF: Windows Workflow in .NET 4*. Apress, June 2010, ISBN: 978-1430227212.

[5] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[6] C. Ellis and G. Nutt, "Modeling and enactment of workflow systems," in *Application and Theory of Petri Nets 1993*, ser. LNCS. Springer, 1993, vol. 691, pp. 1–16.

[7] W. M. P. van der Aalst, "Three Good Reasons for Using a Petri-net-based Workflow Management System," in *Proceedings of the International Working Conference on Information and Process Integration in Enterprises*. Springer, 1996, pp. 179–201, USA, Cambridge, Massachusetts.

[8] N. Lohmann, "A Feature–Complete Petri Net Semantics for WS–BPEL 2.0 and its Compiler BPEL2oWFN," Humboldt–Universität zu Berlin, Informatik-Berichte 212, August 2007.

[9] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, "Formal semantics and analysis of control flow in WS–BPEL," *Science of Computer Programming*, vol. 67, no. 2, pp. 162–198, 2007.

[10] Y. Yang, Q. Tan, J. Yu, and F. Liu, "Transformation BPEL to CP-Nets for Verifying Web Services Composition," in *Proceedings of the International Conference on Next Generation Web Services Practices*. IEEE Computer Society, 2005, pp. 137–142, Korea, Seoul.

[11] C. Gierds, A. J. Mooij, and K. Wolf, "Reducing Adapter Synthesis to Controller Synthesis," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 72–85, 2012.

[12] B. Benatallah, F. Casati, D. Grigori, H. Nezhad, and F. Toumani, "Developing adapters for web services integration," in *Advanced Information Systems Engineering*. Springer, 2005, pp. 415–429, Portugal, Porto.

[13] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *Journal of Systems and Software*, vol. 74, no. 1, pp. 45–54, 2005.

[14] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, "Formalizing web service choreographies," *Electronic Notes in Theoretical Computer Science*, vol. 105, pp. 73–94, 2004.

[15] M. Dumas, M. Spork, and K. Wang, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," in *Proceedings of the 4th International Conference on Business Process Management*. Springer, 2006, pp. 65–80, Austria, Vienna.

[16] H. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 993–1002, Canada, Banff.

[17] W. M. P. van der Aalst, A. Mooij, C. Stahl, and K. Wolf, "Service Interaction: Patterns, Formalization, and Analysis," in *Formal Methods for Web Services*. Springer, 2009, vol. 5569, pp. 42–88.

[18] C. Gierds, A. Mooij, and K. Wolf, "Specifying and generating behavioral service adapters based on transformation rules," Humboldt-Universität zu Berlin, Institut für Informatik, Tech. Rep., 2008.

[19] E. Kindler, "A compositional partial order semantics for Petri net components," in *Application and Theory of Petri Nets 1997*. Springer, 1997, vol. 1248, pp. 235–252.

[20] P. Massuthe, W. Reisig, and K. Schmidt, "An Operating Guideline Approach to the SOA," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 3, pp. 35–43, 2005.

[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*, vol. 4, no. 1, pp. 3–13, 2009.

[22] M. Zapletal, W. M. P. van der Aalst, N. Russell, P. Liegl, and H. Werthner, "An Analysis of Windows Workflow's Control-Flow Expressiveness," in *Proceedings of the 2009 Seventh IEEE European Conference on Web Services*. IEEE, 2009, pp. 200–209, The Netherlands, Eindhoven.

[23] J. Lenhard, A. Schönberger, and G. Wirtz, "Edit Distance-based Pattern Support Assessment of Orchestration Languages ," in *Proceedings of On the Move 2011 Confederated International Conferences: CoopIS, IS, DOA and ODBASE*. Springer, 2011, pp. 137–154, Greece, Crete, Hersonissos.

[24] B. J. F. De Smet, K. Steurbaut, S. Van Hoecke, F. De Turck, and B. Dhoedt, "Dynamic Workflow Instrumentation for Windows Workflow Foundation," in *International Conference on Software Engineering Advances*. IEEE, 2007, France, Cap Esterel.

[25] J. Lenhard, "A Pattern-based Analysis of WS-BPEL and Windows Workflow," Fakultät Wirtschaftsinformatik und Angewandte Informatik, Tech. Rep. 88, 2011.

[26] OASIS, "Universal Business Language Version 2.1 - Committee Specification Draft 02 / Public Review Draft 02," May 2011.