World Wide Web (2011) 14:27–52
DOI 10.1007/s11280-010-0096-y

Implementing process views in the web service environment

Xiaohui Zhao · Chengfei Liu · Wasim Sadiq · Marek Kowalkiewicz · Sira Yongchareon

Received: 8 December 2009 / Revised: 28 June 2010
Accepted: 8 July 2010 / Published online: 20 July 2010

Abstract Web service and business process technologies are widely adopted to facilitate business automation and collaboration. Given the complexity of business processes, it is a sought-after feature to show a business process with different views to cater for the diverse interests, authority levels, etc., of different users. Aiming to implement such flexible process views in the Web service environment, this paper presents a novel framework named FlexView to support view abstraction and concretisation of WS-BPEL processes. In the FlexView framework, a rigorous view model is proposed to specify the dependency and correlation between structural components of process views with emphasis on the characteristics of WS-BPEL, and a set of rules are defined to guarantee the structural consistency between process views during transformations. A set of algorithms are developed to shift the abstraction and concretisation operations to the operational level. A prototype is also implemented for the proof-of-concept purpose.

The work was done while Xiaohui Zhao was working at Swinburne University of Technology, Australia.

X. Zhao
Information Systems Group, Department of Industrial Engineering & Innovation Sciences, Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: x.zhao@tue.nl

C. Liu (✉) · S. Yongchareon
Centre for Complex Software Systems and Services, Swinburne University of Technology, Melbourne, Australia
e-mail: cliu@groupwise.swin.edu.au

S. Yongchareon
e-mail: syongchareon@groupwise.swin.edu.au

W. Sadiq · M. Kowalkiewicz
SAP Research, Brisbane, Australia

W. Sadiq
e-mail: wasim.sadiq@sap.com

M. Kowalkiewicz
e-mail: marek.kowalkiewicz@sap.com

Keywords business process view · web service · workflow management · WS-BPEL

1 Introduction

Service-oriented architecture (SOA) tends to dominate the enterprise scale system infrastructures, where the service components are orchestrated by business processes to collaborate for a mutual goal [1, 11, 21, 22]. As one of the most mature SOA solutions, Web services are widely adopted to realise SOA and the Business Process Execution Language for Web Services (WS-BPEL) [2, 7] is particularly used to specify the business processes for driving Web service composition and coordination.

Recently, the concept of process views has been proposed to generate a partial and temporal representation of a business process. This mechanism provides a better granularity control of business process representation. Further, this feature significantly enhances the flexibility of business process management to adapt to the diverse authority levels and interests of different users. Reluctantly, most of current business process modelling languages, including WS-BPEL, stick to a fixed description of business processes. Although WS-BPEL can define both abstract processes and executable processes, WS-BPEL is in lack of mechanisms to automatically represent a business process at different abstraction levels on demand. This shortcoming obstructs the further adoption of Web services to real enterprise information systems. For example, in an enterprise with strict organisational hierarchy, some roles are only allowed to see part of the business process details at a time [3]. Besides, users with different interests may intend to see different views of the same business process, such as the workshop assistant likes to know the detailed procedure of pipeline operations yet the workshop manager may prefer to see a concise view of the pipeline production process. Further, such requirements may result from the aspect of flexible visualisation of business processes. For users with a smart phone as the business process browser, due to the limit of screen size, they intend to display a reduced version of the business process, and choose the interested area, like a "hot spot", to see the details of that area. A good example is *google maps*, which allows users to freely zoom in or zoom out a map, while the displayed details on map automatically adapt to the scale level, for instance, streets and roads are shown on a large scale map, yet a small scale map only shows suburbs and towns.

Aiming to incorporate these appealing process view features into current process orchestrated Web services applications, this paper proposes a novel FlexView framework to facilitate the process view abstraction and concretisation for WS-BPEL processes. With FlexView, users are allowed to define and switch among the different views of the same WS-BPEL process. This mechanism enables the "smart zooming" function for business process representation, which has been longed for by the practical enterprise applications for flexible visualisation, authority control, privacy protection, process analysis purposes, etc [16, 35]. A comprehensive model defines the structural constructs of a business process and the relations between them. A series of algorithms formally illustrate how to enforce the process abstraction and concretisation operations in compliance with structural consistency.

The remainder of this paper is organised as follows: Section 2 introduces the structural elements of WS-BPEL, and analyses their characteristics; Section 3 discusses the requirements for supporting flexible views with a motivating example; Section 4 presents a process component model with a set of rules on structural consistency and validity, as well as the algorithms for realising abstraction and concretisation; Section 5 addresses the

incorporation of FlexView into WS-BPEL, and also introduces the implementation of a prototype; Section 6 reviews the related work and discusses the advantages of our framework; concluding remarks are given in Section 7 with an indication for the future work.

2 Preliminary of WS-BPEL

Based on the XML syntax, WS-BPEL defines a model and a grammar for describing the behaviours of a business process based on interactions between the process and its partners. The interaction with each partner occurs through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in a partner link. A WS-BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. WS-BPEL also introduces mechanisms for dealing with business exceptions, processing faults and compensations.

The core set of WS-BPEL 1.1 consists of a series of elements as shown in Figure 1. In regard to their functions, these elements can be classified into three categories, namely declaration elements, control flow constructs, and activities. The texts along arrows denote the structural relation between these elements, while the italic texts denote the semantic relations. As the root element, <process> element contains the whole content for a business

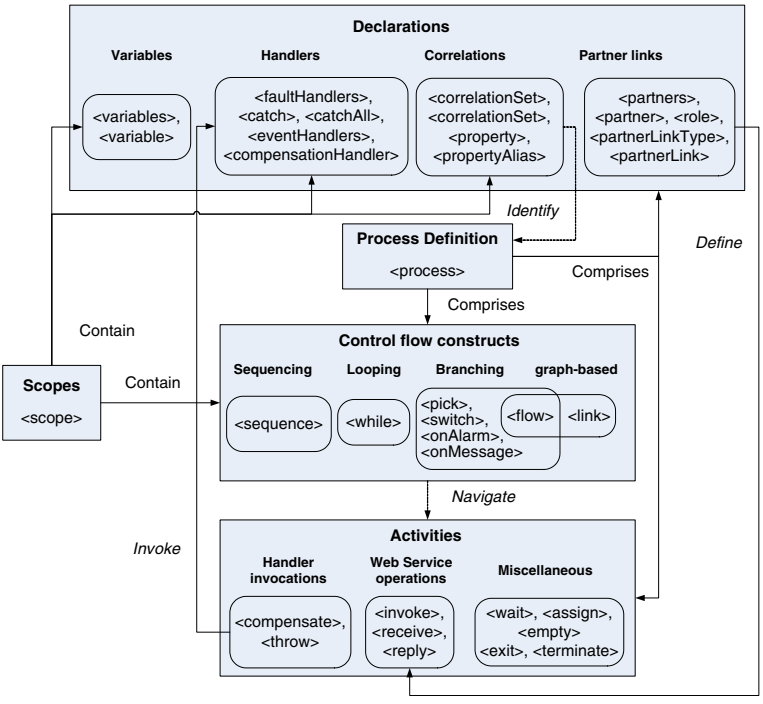

Figure 1 WS-BPEL elements.

process. <scope> element defines the local declarations and exception handlers for a partition of a business process. The other elements are explained as follows:

- The declaration elements include the elements for declaring variables, handlers, correlations and partner links. These components will serve the main business process.
- The control flow constructs include the elements for defining the control flow of a business process. These constructs support standard sequencing, looping, and branching patterns. Particularly, elements <flow> and <link> support the conventional graph-based control flow modelling scheme.
- The activity elements include the elements supporting functional operations of a business process, such as Web service implementation and invocations, handler invocations, etc.

WS-BPEL supports different process modelling ways. As identified in [12], WS-BPEL supports both block-structured process and graph-based process modelling. The processes modelled in different manners are also convertible to each other with specific transformation techniques [12, 20]. Besides, WS-BPEL can define inline compensation/fault handlers for <invoke> element, in addition to explicitly using an immediate enclosing scope. To avoid redundancy, we focus exclusively on the block-structured processes in this paper. The details for transforming graph-based processes into block-structured processes can be found in [12, 20].

According to above discussion, we summarise the following features of WS-BPEL process modelling:

1. The tasks on the branches in a <flow> element representing an And-Split/Join structure can be synchronised via a <link> element.
2. In scope c, a <compensate> element can explicitly invoke a compensation handler that is defined in c or the scopes that are enclosed in c.
3. A fault handler can capture and handle faults that are occurred in the same scope. When a fault handler captures a fault, it might throw the same or a different fault to its outer scope using <throw> element.
4. In a scope, the compensation and fault handling can be treated as separate processes from the normal process.

These features characterise the control flow construction and composition in WS-BPEL. This paper will particularly take into account these features in the context of business process view transformation.

3 Motivating example

In the Web service application environment, WS-BPEL has been widely adopted and supported by many leading software companies. Figure 2 shows some WS-BPEL diagrams drawn with SAP's business process modelling tool—Maestro for BPEL. These diagrams represent different process views for a simplified sales management process. Figure 2(a) show the original content of this process, i.e., the base process, where the process starts from receiving purchase orders, and then handles the production, cost analysis and shipping planning concurrently, and finally terminates by sending the invoice. Each task may interact with proper Web service(s) though WS-BPEL <invoke> or <receive> activity to fulfil the assigned mission. The dashed arrows denote the synchronisation dependencies between tasks, which are represented by <link> activities in WS-BPEL. For example, the arrow

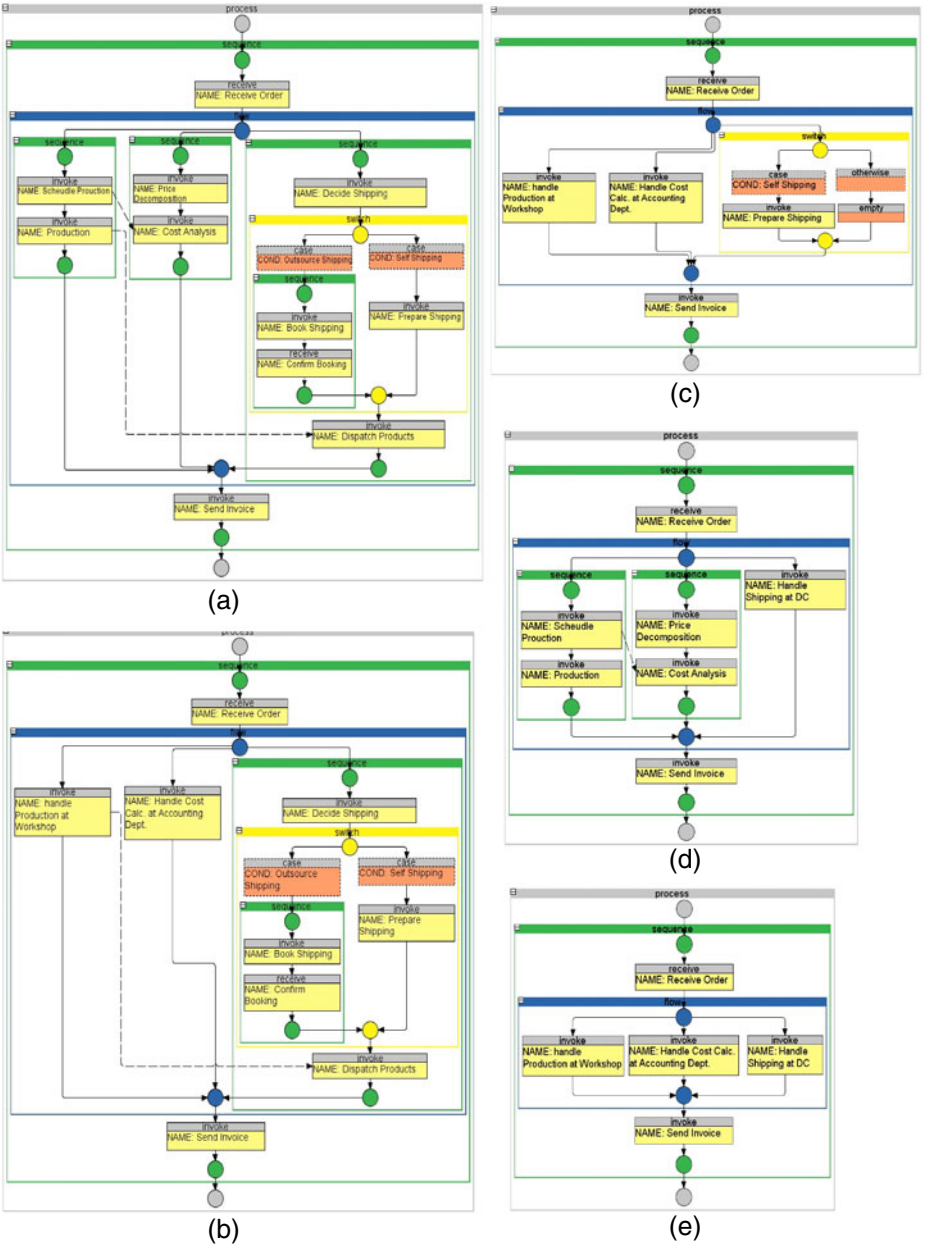

Figure 2 Motivating example business process and process views.

between "production" and "dispatch products" denotes that task "dispatch products" can only start after the completion of "production".

Four departments are involved in this business process, viz., sales department, workshop, accounting department, and distribution centre. The users of different departments may have different authorities or interests to observe the business process. For

example, a user from the distribution centre may only care about the shipping details. Thus, the user may choose to "zoom out" the details for production and cost handling from the original business process, and thereby the user can obtain a process view for this business process as shown in Figure 2(b). In this view, the details for production and cost handling are abstracted into two new tasks, i.e., "handle production at workshop" and "handle cost calculation at accounting department". These two tasks hide the details yet preserve the existence of the production and cost handling procedures. In this transformation, the related links are hidden automatically, as well as the synchronisation link from "schedule production" to "cost analysis". The synchronisation link from "production" to "dispatch products" is converted to connect "handle production at workshop" to "dispatch products", as these two tasks inherit the synchronisation dependency of the former one.

Suppose a user is using a hand-held device to view this business process. Due to the limited screen size, the user may prefer to display the details of a single shipping option at a time. In this case, the representation of this business process can be transformed to the view shown in Figure 2(c), where the shipping procedure is represented as an Or-split/join structure. One branch of this Or-split/join structure contains task "prepare shipping", and the other branch is only with an empty task, which indicates that there is an alternative shipping option besides self-shipping. The user may later on choose to "zoom in" this empty task to see the details of the alternative option.

Some junior staff of other departments may not be authorised to see the shipping details. Therefore, a process view like Figure 2(d) can be provided to them, where all the shipping details are hidden in a new task "handle shipping at distribution centre". In this hiding behaviour, the synchronisation link from "production" to "dispatch products" is also hidden, because the underlying synchronisation dependency is already not effective for this view. Figure 2(e) displays a further abstracted view of the business process, which only outlines the core part of the business process with three parallel tasks. The authorised users can choose to concretise the interested part to see more details.

Such representation mechanism separates the process representation from the process execution, and thereby can provide highly flexible views over the underlying business process. This capability brings benefits in aspects of process visualisation, authority control, privacy protection, service advertising, etc., for business process management.

This example reveals the demand for flexible process representations in the practical environment. To realise the adjustability and customisability of the process representation granularity, new mechanisms are on demand to support process abstraction and concretisation functions on the fly. In details, we summarise the technical requirements as follows:

Maintain the correspondence between the hidden process fragments and the corresponding tasks/links, and therefore enable wrapping a process fragment into a specific task or link, and releasing the process fragment back from a task or link.

- Preserve the structural information of a business process, such as the execution order between tasks, split/join structures, etc., during process view abstraction/concretisation operations.
- Preserve the synchronisation links, and hide, reveal or relocate them properly during process view abstraction/concretisation operations.
- Support cascading abstraction/concretisation operations.
- Guarantee the structural consistency and validity of process views during transformations.

For the first requirement, our FlexView framework employs a process component model to describe the structure of process views and structural components, and maintain the relations between structural components. For the remaining requirements, we define a set of rules regulate the structural consistency during the view transformations. A series of algorithms are designed to enforce the process view abstraction and concretisation. A prototype is developed for the proof-of-concept purpose. The reported work in this paper is based on a preliminary version of our work on process view abstraction and concretisation [36], with significant improvements and extensions on theoretical analysis, algorithms and system implementation.

4 FlexView framework

4.1 Process component model

Based on the analysis of WS-BPEL characteristics and the requirements for flexible process representation, we define a process component model in this section. This model well describes the structure of business processes and process views, and maintains the correspondence between related structural components.

Definition 1. *(Gateway)* In this model, gateways are defined as the dedicated control flow constructs for a business process.

According to the main WS-BPEL structured activities, we define five types of gateways, namely *Or-Split*, *Or-Join*, *And-Split*, *And-Join*, and *Loop* to represent the structure of a control flow. These gateway types correspond to the WS-BPEL structured activities, <flow> and </flow>, <switch> and </switch>, and <while>, respectively. Figure 3 shows the samples of these gateways, respectively. *Or-Split/Join* and *Loop* gateways may attach conditions to restrict the control flow.

Definition 2. *(Synchronisation Link)* Synchronisation links denote the links between tasks on different branches of an And-Split/Join structure. Synchronisation links own a higher priority than normal links in terms of execution order.

In WS-BPEL, the <link> element can be used to denote the synchronisation dependency between the tasks of different branches in an *And-Split/Join* structure. In our model, we use a synchronisation link to represent such synchronisation dependency. For example, in Figure 3(b), the synchronisation link between t_i and t_j, represented as a dashed arrow, denotes that t_j can only start after the completion of t_i.

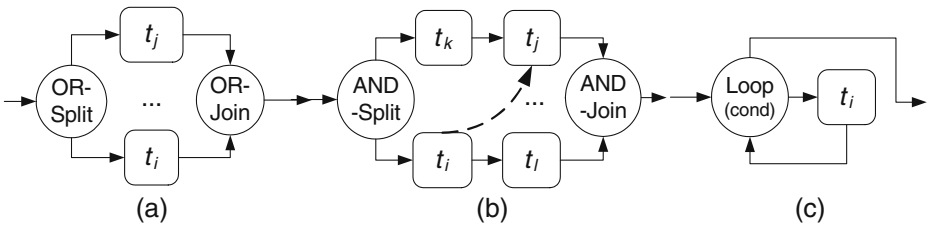

Figure 3 Gateway examples.

Definition 3. *(Process Fragment)* A process fragment denotes a consecutive part of a business process. A process fragment $s = (N, G, E, L)$ consists of

- a set $N = \{n_1, n_2, \ldots, n_x\}$ of tasks, where $n_i \in N$ $(1 \le i \le x)$ represents a task of s.
- a set $G = \{g_1, g_2, \ldots, g_y\}$ of gateways, where $g_i \in G$ $(1 \le i \le y)$ represents a gateway of s.
- a set E of directed edges. An edge $e = (m_1, m_2) \in E$ corresponds to the control dependency between m_1 and m_2, where $m_1 \in N \cup G$, $m_2 \in N \cup G$.
- a set L of synchronisation links. A synchronisation link $l = (m_1, m_2) \in L$ corresponds to the synchronisation dependency between m_1 and m_2, where $m_1 \in N \cup G$, $m_2 \in N \cup G$.

where each element $g \in G$ has exactly one gateway type. Let $type$: $G \rightarrow \{Loop, And\text{-}Join, And\text{-}Split, Or\text{-}Join, Or\text{-}Split\}$ be a function to return the type of gateway.

Functions $ind(m)$ and $outd(m)$ define the number of edges which take m as the target node and the source node, respectively. Note, *ind* and *outd* only count the number of edges but not synchronisation links.

According to the natural characteristics of these gateways, we can define the following rules in terms of the incoming and outgoing degrees:

if $type(g)$ = "*Loop*"	$ind(g)=1, outd(g)=2$	If g is the starting node of a process;
	N/A	A *loop* gateway is not allowed to be the ending node of a process;
	$ind(g)=2, outd(g)=2$	Otherwise.
if $type(g)$ = "*And-Split*" or "*Or-Split*"	$ind(g)=0, outd(g)>1$	If g is the starting node of a process;
	N/A	A split gateway g is not allowed to be the ending node of a process;
	$ind(g)=1, outd(g)>1$	Otherwise.
if $type(g)$ = "*And-Join*" or "*Or-Join*"	*N/A*	A join gateway is not allowed to be the starting node of a process;
	$ind(g)>1, outd(g)=0$	If g is the ending node of a process;
	$ind(g)>1, outd(g)=1$	*Otherwise*.

Definition 4. *(Single-Entry-and-Single-Exit (SESE) Process Fragment)* An SESE process fragment denotes a process fragment which has only one entry node and one exit node. An SESE process fragment $se = (N, G, E, L, m_s, m_t)$ consists of

- N, G, E, L are defined as same as in process fragment;
- $m_s \in N \cup G$ is the starting node of se that $ind(m_s) = 0$.
- $m_t \in N \cup G$ is the terminating node of se that $outd(m_t) = 0$.

Definition 5. *(Scoped Process Fragment)* A scoped process fragment denotes an SESE process fragment with optional compensation and fault handling process(es). A scoped process fragment $sp = (N, G, E, L, m_s, m_t, L_0, ch, fh)$ consists of

- N, G, E, L, m_s, and m_t are defined as same as in SESE process fragment;
- L_0 is a set of hidden synchronisation links, i.e., the synchronisation links that have a node not included in N or G. $\forall l = (m_1, m_2) \in L_0$, $(m_1 \in N \cup G) \wedge (m_2 \notin N \cup G)$ or $(m_1 \notin N \cup G) \wedge (m_2 \in N \cup G)$. The synchronisation links in L_0 are not displayable as they connect to foreign nodes, but such synchronisation link information is preserved for process fragment composition.

– *ch* denotes a set of optional compensation handling processes, each of which corresponds to an SESE process fragments;
– *fh* denotes a set of optional fault handling processes, each of which corresponds to a process fragment;

In this paper, we confine that the process view manipulation is only conducted over scoped process fragments. A *business process* may contain multiple scoped process fragments, and the definition of business process itself also complies with the definition of scoped process fragment. The relation between a business process and its constitute scoped process fragments is reserved in the process hierarchy.

Definition 6. *(Process Hierarchy)* A process hierarchy $\Gamma(p)$ for business process p maintains all the related scoped process fragments and the mapping information for process view generation. $\Gamma(p)$ can be represented as tuple $(S, \delta, \gamma, \lambda)$ where

– S is a finite set of distinct scoped process fragments.

For a scoped process fragment $sp \in S$, $\forall l = (n_1, n_2) \in sp.L \cup sp.L_0$ $\exists sp_1 \in S$ $(n_1 \in sp_1.N \cup sp_1.G) \wedge (n_2 \in sp_1.N \cup sp_1.G)$, or $\exists sp_1 \in S$, $sp_2 \in S$, $(n_1 \in sp_1.N \cup sp_1.G) \wedge (n_2 \in sp_2.N \cup sp_2.G)$.

– $sp_0 \in S$ is the root scoped process fragment, which shows the most abstracted view of p.
– δ: $E' \rightarrow S'$ $(E' \subseteq \bigcup_{sp \in S} sp.E$ and $S' \subseteq S \backslash \{sp_0\})$ is a bijection describing the relation between edges and scoped process fragments. Correspondingly, we have the inverse function δ^{-1}: $S' \rightarrow E'$.
– γ: $N' \rightarrow S'$ $(N' \subseteq \bigcup_{sp \in S} sp.N$ and $S' \subseteq S \backslash \{sp_0\})$ is a bijection describing the relation between nodes and scoped process fragments. Correspondingly, we have the inverse function γ^{-1}: $S' \rightarrow N'$.
– $\lambda T' \rightarrow RCH$ $(T' \subseteq \bigcup_{sp \in S} sp.T$ and $RCH' \subseteq \bigcup_{sp \in S} sp.ch)$ is a bijection describing the relation between compensation invoking tasks and the corresponding compensation handlers.

A scoped process fragment is restricted to occur in only one of the two inverse functions. This denotes that $\forall sp \in S \backslash \{sp_0\}$, if $\delta^{-1}(sp) \neq$ null then $\gamma^{-1}(sp) =$ null; if $\gamma^{-1}(sp) \neq$ null then $\delta^{-1}(sp) =$ null.

The scoped process fragments of a process hierarchy can be defined in a nested manner. As shown in Figure 4, a process hierarchy may contain five scoped process fragments sp_0,

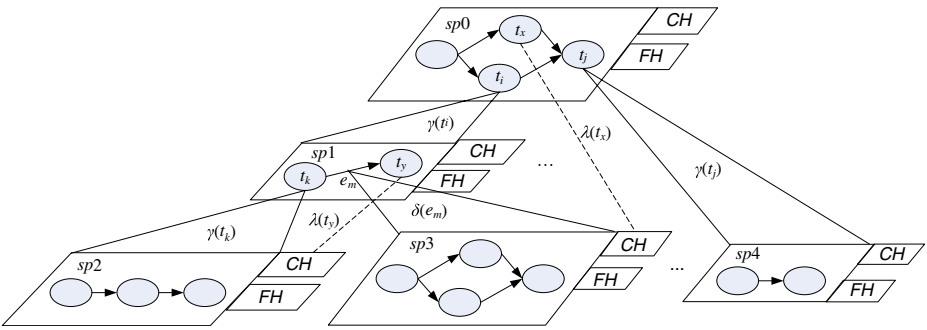

Figure 4 Process hierarchy example.

…, sp_4, and sp_0 is the root scoped process fragment. Shapes labelled "CH" and "FH" denote the attached compensation and fault handling processes.

In this hierarchy, tasks t_i and t_j of scoped process fragment sp_0 correspond to process fragments sp_1 and sp_4, respectively, and therefore t_i and t_j can be mapped to sp_1 and sp_4 by functions $\gamma(t_i)$ and $\gamma(t_j)$. Further, task t_k and edge e_m of sp_1 can be mapped to process fragments sp_2 and sp_3 by functions $\gamma(t_k)$ and $\delta(e_m)$, respectively. In this manner, a process view *concretisation* operation is equivalent to extending a scoped process fragment by replacing a task or edge with the corresponding scoped process fragment. For example, root scoped process fragment sp_1 can be concretised into sp_0+sp_1, sp_0+sp_4, or sp_0+sp_1+sp_4, where tasks t_i and t_j are replaced by the corresponding scoped process fragments. Similarly, sp_0 can be further concretised into sp_0+sp_1+sp_2, sp_0+sp_1+sp_3, sp_0+sp_1+sp_3+sp_4, etc., by replacing the proper task(s) or edge(s) with corresponding scoped process fragments. Adversely, the *abstraction* operation is equivalent to unfolding a scoped process fragment back into a task or edge with functions γ^{-1} and δ^{-1}. For example, the combination of sp_0+sp_1 or sp_0+sp_4 can be abstracted into sp_0 by hiding the scoped process fragment sp_1 or sp_4 into task t_i or t_j, respectively. Therefore, we can see that each result combination denotes a partial view of the business process. Such a process hierarchy is fully customisable, and thereby we can apply different process fragment partitioning and categorising strategies to realise diverse WS-BPEL abstraction schemes according to users' requirements.

Definition 7. *(Process View)* A process view represents a partial and temporal view of a business process. For a given process hierarchy $\Gamma(p)$, a process view v corresponds to a sub tree including the root scoped process fragment, where the mapped tasks/edges are concretised with corresponding scoped process fragments. The structure of a process view v also corresponds to an extended directed graph like a scoped process fragment, but without any hidden content. Therefore, we define the structure of a process view in the form of tuple (N, G, E, L, m_s, m_t, ch, fh), where the constitute sets and elements have the same meanings with those for a scoped process fragment.

A fully concretised process view, i.e., the view containing all the scoped process fragments in this hierarchy, is equivalent to the base business process, while the most abstract process view is the root scoped process fragment itself. The mapping between view components and WS-BPEL elements will be detailed in Section 5.

4.2 Structural consistency and validity rules

To guarantee the structural consistency between the generated process views, this section defines a series of rules to regulate the view transformation in terms of execution order, branch correspondence and synchronisation dependency. Besides, some rules are also defined to check the structural validity on split/join structures.

- *Preliminary*

– A *dummy branch* denotes a branch in a split/join structure such that the branch contains nothing but only one edge.
– A common split gateway predecessor (CSP), x, of a set of tasks, T, denotes a split gateway such that x is the predecessor of each task in T.
– *before*(t_1, t_2) denotes that task t_1 will be executed earlier than task t_2. This means that there exists a path from starting t_1 to t_2 in the corresponding directed graph, while the

path does not contain any synchronisation links if an And-split/join structure or any go-back edge of a loop structure. Apparently, *before* is a transitive binary relation.
- $CSP(t_1, t_2)$ returns the set of common split gateway predecessors of t_1 and t_2, or returns null if the two tasks have no common split gateway predecessors.
- $branch(g, t_1, t_2)$ is a boolean function, which returns true if t_1 and t_2 lie in the same branch led from split gateway g, otherwise returns false.

• *Structural Consistency and Validity Rules*

In regard to an abstraction/concretisation operation, the original process view v_1 and the result view v_2 are required to comply with the following restrictions in terms of structural consistency and validity:

Rule 1. (Order preservation) As for the tasks belonging to v_1 and v_2, the execution sequences of these tasks should be consistent, i.e.,

If $t_1, t_2 \in v_1.N \cap v_2.N$ such that $before(t_1, t_2)$ exists in v_1, then $before(t_1, t_2)$ also exists in v_2.

Rule 2. (Branch preservation) As for the tasks belonging to v_1 and v_2, the branch subjection relationship of these tasks should be consistent, i.e.,

If $t_1, t_2 \in v_1.N \cap v_2.N$ and $g \in CSP(t_1, t_2)$ exist in v_1 and $g \in CSP(t_1, t_2)$ exists in v_2 such that $branch(g, t_1, t_2)$ (or $\neg branch(g, t_1, t_2)$) in v_1, then $branch(g, t_1, t_2)$ (or $\neg branch(g, t_1, t_2)$) in v_2.

These two rules ensure the basic structural consistency between process views, which reflect a kind of observational equivalence to process users, i.e., the original process and its transformed process view represent the same interaction pattern with users. However, this observational equivalence is subject to the constrained process perception, and therefore it corresponds to the weak bi-simulation in the process algebra context. To justify this property of process views, we are to prove that the task hiding and aggregation operations both support this property using the concept of labelled transition system in the process algebra context. The definitions of labelled transition system and weak bi-simulation can be found in Appendix.

(1) Weak simulation in task hiding.

Proof Figure 5(a) gives two BPEL process views, where view v_1 shows the result view of hiding task "Arrange Shipping" of view v_2. This hiding operation can be generalised into the process view transformation of abstracting a scoped process fragment *sp* of v_2 into edge e_0 of v_1, as shown in Figure 5(b). We first transfer v_1 and v_2 into labelled transition systems LTS_1 and LTS_2 as shown in Figure 5(c), respectively. Note, in labelled transition system, edges represent tasks (actions) and nodes represent process states. The tasks included in *sp* in v_2 are changed to τ actions in LTS_2. In LTS_2, τ^* denotes none or a series of invisible transitions that are included in *sp*'s tasks.

For relation $R = \{(m_1, ms)\}$, we have $(P \stackrel{def}{=} n_2) \approx (Q \stackrel{def}{=} \tau^*.n_2)$. Given $P \approx Q$, we can prove that $n_1.P \approx n_1.Q$ using Milner's τ-laws. This will lead to the conclusion of $LTS_1 \approx LTS_2$. □

(2) Weak simulation in task aggregation.

Proof In Figure 6(a), view v_1 shows the result view of aggregating the tasks belonging to the switch structure of view v_2 into a new task "Arrange Shipping". Generally, this transformation is represented as aggregating v_2's scoped process fragment *sp* into a new task *tx*, as shown in Figure 6(b). The underlying semantic is that these tasks are aggregated

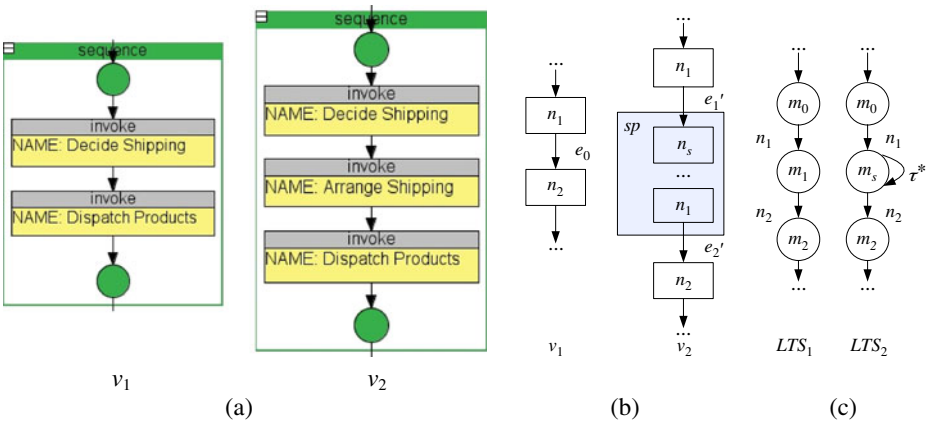

Figure 5 The labelled transition systems for task hiding.

into a delegated one, while the details are hidden. Therefore, LTS_1 can be changed to LTS_2, where states ms and ms' have self-directed τ actions, respectively.

Given relation $R = \{(m_1, ms), (m_2, ms')\}$, we can see that $(P \stackrel{def}{=} tx) \approx (Q \stackrel{def}{=} \tau^*.0.\tau^*)$ according to the definition of weak simulation (please refer to Appendix). Further, we can conclude that $LTS_1 \approx LTS_2$. □

Rule 3. (Synchronisation dependency preservation) If an abstraction operation involves any tasks with synchronisation links, the synchronisation links should be rearranged to preserve the synchronisation dependency. Assume that scoped process fragment *sp* comprising tasks t_1 and t_2 is to be abstracted into a compound task t_c as shown in Figure 7,

– for task $t_x \in s.N$ and t_x has an outgoing synchronisation link l,

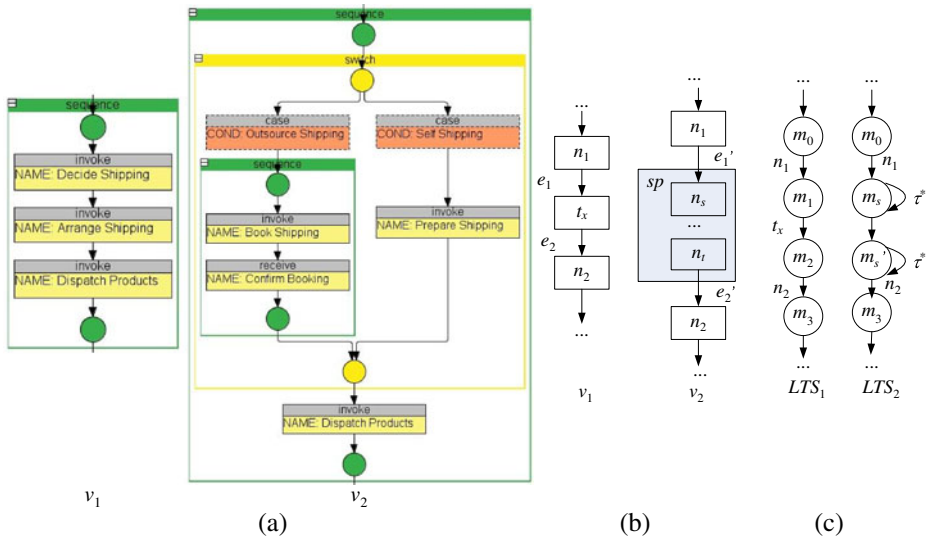

Figure 6 The labelled transition systems for task aggregation.

If $\forall t \in sp.N$, $before(t, t_x)$ then the source task of l should be changed to t_c, otherwise l should be hidden.

– for task $t_x \in sp.N$ and t_x has an incoming synchronisation link l,

If $\forall t \in sp.N$, $\neg before(t, t_x)$ then the target task of l should be changed to t_c, otherwise l should be hidden.

In Figure 7, the transformation from (a1) to (a2), where t_1 and t_2 are hidden in task t_c, and the transformation from (a1) to (a3), where t_4 and t_5 are hidden in task t_c, illustrate the two mentioned scenarios, respectively.

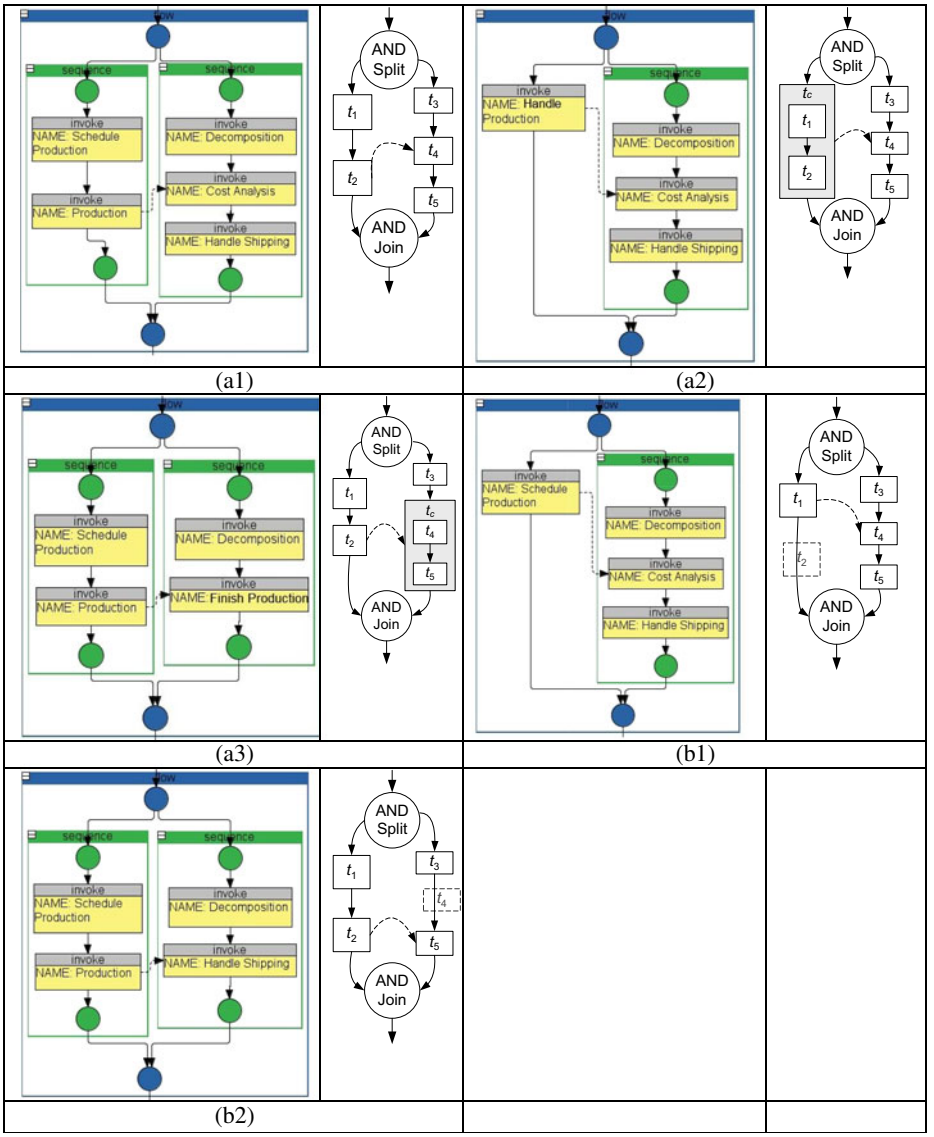

Figure 7 Synchronisation link handling.

In the case that a task involving a synchronisation link is abstracted into an edge, the re-arrangement of synchronisation links is subject to Rule 1. For example, Figure 7(b1) and (b2) illustrate the re-arrangements in cases that t_2 and t_4 are hidden in edges, respectively.

Rule 4. (No empty Split/Join or Loop structures) If a loop structure contains no tasks, or if a split/join structure contains only dummy branches, then the loop or split/join structure should be hidden.

Rule 5. (No dummy or single branch in And-Split/Join structures) If an And-split/join structure contains both dummy and non-dummy branches, then the dummy branch(es) should be hidden. If the And-split/join structure contains only one non-dummy branch, then the And-split/join structure will be degraded into a sequential structure.

As shown in Figure 8(a1), when tasks t_3 and t_4 are hidden, the And-split/join structure will decay into a sequential structure as shown in Figure 8(a2).

Rule 6. (Dummy branch in Or-Split/Join structures) If an Or-split/join structure contains a dummy branch, then this dummy branch should remain to indicate the existence of an alternative execution path. If an Or-split/join structure contains multiple dummy branches, these branches should merge into one dummy branch.

As shown in Figure 8(b1), when tasks t_3 and t_4 are hidden, the Or-split/join structure will use an empty task to hold the place for that dummy branch, as shown in Figure 8(b2). This is to preserve the possibility that branch containing t_1 and t_2 may be bypassed during execution.

Rule 7. (Compensation Handler Redirection) When a scoped process fragment *sp* is hidden, any reference to *sp*'s compensation handler(s) should be redirected to a virtual handler to indicate the indivisibility of the referenced handler(s). When *sp* is released during a concretisation operation, the redirected reference(s) should be changed back to *sp*'s compensation handler(s).

Fault handlers cannot be explicitly invoked, and therefore they do not need such redirection.

4.3 Enforcing process abstraction and concretisation

To realise the view abstraction and concretisation in compliance with the defined rules, we developed a series of algorithms to formalise the process view transformations.

Given a process hierarchy $\Gamma(p) = (S, \delta, \gamma, \lambda)$, the following basic functions are to be used in the algorithms: *elementType*(y) returns whether y is a task or an edge. *combineSProc*(sp_1, sp_2) returns the result scoped process fragment by combining the constitute sets, i.e., N, G, E, L, L_0, *ch* and *fh* of scoped process fragments sp_1 and sp_2 together. *removeSProc*(sp_1, sp_2) returns the result scoped process fragment by removing the constitute sets of scoped process fragment sp_2 from scoped process fragment sp_1. *toSequence*(sp, g_1, g_2) returns the result scoped process fragment by flating a single branch split/join structure scoped by gateways g_1 and g_2 in scoped process fragment *sp* into a sequence structure, i.e., removes the two gateways and re-connects the gateways' adjacent nodes to the single branch. *spSet*(S, x) returns the scoped process fragments that are in set S of process hierarchy and contain task or edge x.

To tackle the modification of synchronisation links during process view transformations according to Rule 3, we develop a general algorithm *adjustSyncLinks* to sort out the influenced synchronisation links for both process view abstraction and concretisation operations.

Algorithm. adjustSyncLinks(*sp*, *zoom*, *sp*°, *y*)

Input *sp* – a scoped process fragment;
zoom – a string indicating whether it is for a concretisation or abstraction operation;
sp° – the scoped process fragment to be zoomed in from a task/edge during a contraction operation, or to be zoomed out during an abstraction operation;
y – the involved task or edge during the process view transformation;
Output *sp′* – the result scoped process fragment after structuring.

```
1  if zoom="zoomIn" then
2    for each sync link l=(m1, m2)∈ sp.L0∪sp°.L0
3      if (m1∈sp′.N∪sp′.G)∧(m2∈sp′.N∪sp′.G) then
4          sp′.L= sp′.L ∪ { l };   sp′.L0=sp′.L0\{ l };
5      end if
6    for each sync link l=(m1, m2)∈sp′.L\sp.L
7      if (m1=y) or (m2=y) then sp′.L= sp′.L \{ l };
8    for each sync link l=(m1, m2)∈sp′.L0\sp°.L0
9      if (m1=y) or (m2=y) then sp′.L0=sp′.L0\{ l };
10   return sp′;
11 else
12   sp′=removeSProc(sp, sp°);
13   if elementType(y)=task then
14     for each sync link l=(m1, m2)∈sp′.L
15       if (m1∈s°p.N∪sp°.G) and (∀t∈sp°.N, before(t, m1)) then
16         sp′.L= sp′.L ∪ { (y, m2) }; sp′.L= sp′.L \ { l };
17       else if (m2∈sp°.N∪sp°.G) and (∀t∈sp°.N, ¬before(t, m2)) then
18         sp′.L= sp′.L ∪ { (m1, y) }; sp′.L= sp′.L \ { l };
19       end if
20   if elementType(y)=edge then
21     for each sync link l=(m1, m2)∈sp′.L
22       if (m1∈sp°.N∪sp°.G) and (∀t∈sp°.N, before(t, m1)) then
23         suppose y =(m3, m4); sp′.L= sp′.L∪{ (m3, m2) }; sp′.L= sp′.L\{l};
24       else if (m2∈sp°.N∪sp°.G) and (∀t∈sp°.N, ¬before(t, m2)) then
25         suppose y=(m3, m4); sp′.L= sp′.L∪{ (m1, m4) }; sp′.L= sp′.L\{l};
26       end if
27     return sp′;
28 end if
```

In case of a process view concretisation operation, lines 2–5 check whether any hidden synchronisation links turn visible during the process view concretisation. If the concretisation operation zooms in from a specific task, i.e., variable *y* is not null and *zoom* has value of "zoomIn", lines 6–7 delete the synchronisation links that are involved with the to-be-zoomed task, because the newly revealed synchronisation links from *sp*° will replaces these links, while lines 8–9 adjust the hidden synchronisation links that are involved with the to-be-zoomed task.

Lines 11–27 handle the synchronisation links for a process view abstraction operation. If it is to abstract a scoped process fragment into a task, lines 14–16 and lines 17–19 rearrange the synchronisation links to preserve the synchronisation dependency in cases that *sp*° has an incoming link and the link leaves from the last node or joins to the first node of *sp*°, respectively. Similarly, lines 20–26 handle the same cases that the operation is to abstract a scoped process fragment into an edge. The time complexity is $o(n^2)$, where n is the number of tasks and gateways belonging to the scoped process fragment.

Algorithm *edgeZoomIn* is responsible for concretising a scoped process fragment by extending an edge.

Algorithm. edgeZoomIn(*sp*, *e*)

Input	*sp*	–	a scoped process fragment;
	e	–	an edge of scoped process fragment *sp*;
Output	*sp′*	–	the result scoped process fragment after concretising *e*.

```
1  sp′=sp;
2  let e=(m_a, m_b);
3  if δ(e)=null then return sp else sp°=δ(e);
4  sp′.E= sp′.E ∪ { (m_a, sp°.m_s) };
5  sp′.E= sp′.E ∪ { (sp°.m_t, m_b) };
6  sp′.E= sp′.E \{ e };
7  sp′=combineSProc(sp′, sp°);
8  sp′=adjustSyncLinks(sp′, "zoomIn", sp°, null);
9  for each t∈sp′.T
10     if t is a compensation invoking task then
11        if t refers to "virtual compensation handler" and λ(t)∈sp°.ch then change t to refer to
          λ(t).
12 return sp′;
```

Line 4 connects edges according to Rule1 and Rule 2. Lines 6–7 replace edge *e* with the mapped scoped process fragment. Line 8 calls *adjustSyncLinks* algorithm to sort out the affected synchronisation links. Lines 9–11 change the references to virtual compensation handler back to the real compensation handlers, if these handlers are revealed when extending *sp*, according to Rule 7. The time complexity is $O(n)$, where n is the number of tasks belonging to the scoped process fragment.

Algorithm *taskZoomIn* is responsible for concretising a scoped process fragment by extending a specific task.

Algorithm. taskZoomIn(*sp*, *t*).

Input	*sp*	–	a scoped process fragment;
	t	–	a task of scoped process fragment *sp*;
Output	*sp′*	–	the result scoped process fragment after concretising *t*.

```
1  sp′=sp;
2  if γ(t)=null then return sp else sp°=γ(t);
3  if t=sp′.m_s then sp.m_s=sp°.m_s;
4  if t=sp′.m_t then sp.m_t=sp°.m_t;
5  do while (∃e=(m_x, t)∈sp′.E)
6     sp′.E= sp′.E \{ e };
7     sp′.E= sp′.E ∪ { (m_x, sp°.m_s) };
8  loop
9  do while (∃e=(t, m_y)∈sp′.E)
10    sp′.E= sp′.E \{ e };
11    sp′.E= sp′.E ∪ { (sp°.m_t, m_y)};
12 loop
13 sp′.T= sp′.T \{ t };
14 sp′=combineSProc(sp′, sp°);
15 sp′= adjustSyncLinks( sp′, "zoomIn", sp°, t );
16 for each t∈sp′.T
17    if t is a compensation invoking task then
18       if t refers to "virtual compensation handler" and λ(t)∈sp°.ch then change t to refer to
         λ(t).
19 return sp′;
```

Lines 3–4 handle the connection in case that the task to concretise is the starting or ending node. Lines 5–12 connect edges according to Rule 1 and Rule 2. Lines 13–14 replace task t with scoped process fragment sp. Line 15 calls *adjustSyncLinks* algorithm to sort out the affected synchronisation links. Lines 16–18 redirect the references to virtual compensation handler back to the real compensation handlers. The time complexity is $O(n)$, where n is the number of tasks belonging to the scoped process fragment.

Algorithm *taskZoomOut* is responsible for abstracting a scoped process fragment sp by hiding a part of it into the corresponding edge or task.

Algorithm. taskZoomOut(sp, x)

	Input	sp –	a scoped process fragment;
		x –	a task or edge of scoped process fragment sp;
	Output	sp' –	the result scoped process fragment after abstracting the part containing x.
1	$sp'=sp$;		
2	**if** $spSet(S, x)$=null **then return** sp **else** $sp^{\circ}=spSet(S, x)$;		
3	**if** $\delta^{-1}(sp^{\circ})\neq$null **then**		
4	$sp'.E= sp'.E \cup \{ \delta^{-1}(sp^{\circ}) \}$; $y= \delta^{-1}(sp^{\circ})$;		
5	**else if** $\gamma^{-1}(sp^{\circ})\neq$null **then**		
6	$sp'.T= sp'.T \cup \{ \gamma^{-1}(sp^{\circ}) \}$; $y= \gamma^{-1}(sp^{\circ})$;		
7	**end if**		
8	$sp'= adjustSyncLinks(sp'$, "zoomOut", $sp^{\circ}, x)$;		
9	$sp'=removeSProc(sp', sp^{\circ})$;		
10	**do**		
11	**for each** loop structure with loop gateway g in sp'		
12	**if** $\exists e=(g, g)\in sp'.E$ **then** $sp'=removeLoop(sp', g)$;		
	/* remove empty loop structure */		
13	**for each** split/join structure scoped by split gateway g_1 and join gateway g_2, in sp'		
14	$flag$=0;		
15	**if** $(outd(g_1)=ind(g_2)=1)$ and $(\exists e=(g_1, g_2)\in sp'.E)$ **then**		
16	$sp'.E= sp'.E \setminus \{ e \}$; $sp'=toSequence(sp', g_1, g_2)$; $flag$=1;		
17	**end if**		
18	**if** $(sp'.type(g_1)$=And-split) AND $(\exists e=(g_1, g_2)\in sp'.E)$ **then** $sp'.E= sp'.E \setminus \{ e \}$;		
19	**if** $outd(g_1)=ind(g_2)=1$ **then**		
20	$sp'= toSeqence(sp', g_1, g_2)$; $flag$=1;		
21	**end if**		
22	**end for**		
23	**loop until** ($flag$=0)		
24	**for each** $t\in sp'.T$		
25	**if** t is a compensation invoking task **then**		
26	**if** $\lambda(t)\notin sp'.ch$ **then** change t to refer to "virtual compensation handler"		
27	**return** sp';		

Lines 3–7 replace the proper scoped process fragment with the mapped task or edge. Line 8 calls *adjustSyncLinks* algorithm to sort out the affected synchronisation links. Lines 10–23 iteratively check the structural consistency according to Rules 4–6, until no conflicts exist. According to Rule 4, lines 11–12 and lines 13–17 delete empty loop structures and empty split/join structures, respectively. According to Rules 5 and 6, line 18 deletes dummy branches in an And-split/join structure, and lines 19–21 flat any split/join structures with single branches into sequential structures. Note, due to the set definition, the dummy branches in an Or-split/join structure are already combined together. Lines redirect the references to the compensation handlers which are hidden during the abstraction, to a virtual compensation handler. The time complexity is $O(max$

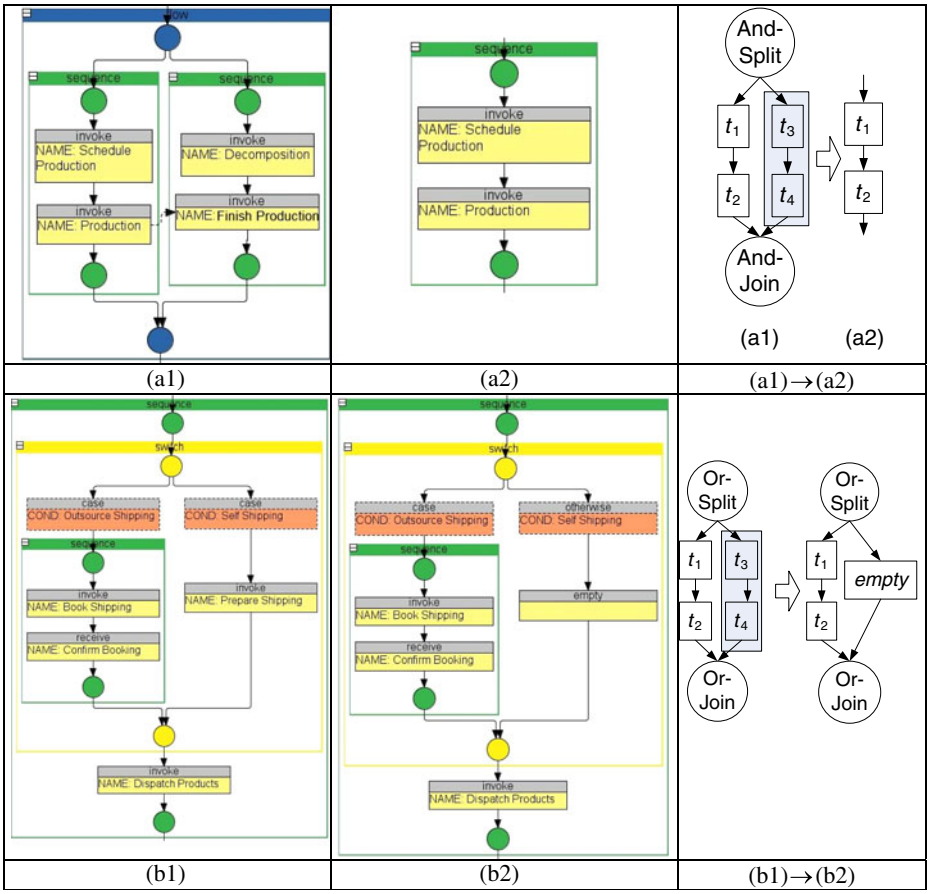

Figure 8 Branch handling examples.

(m^2, n)), where m and n are the numbers of gateways and tasks belonging to the scoped process fragment, respectively.

The result scoped process fragment from these algorithms can be easily converted to a process view for representation, by discarding set L_0.

5 Incorporation into WS-BPEL

5.1 Mapping to WS-BPEL

To incorporate the proposed process view framework into the WS-BPEL processes, we first need to combine the process component model with WS-BPEL model. According to the structural characteristics, Table 1 lists the correspondences between the constructs of our model and WS-BPEL elements.

Besides, in WS-BPEL, every edge is implicitly represented, i.e., the execution sequence is determined by the occurrence sequence of elements nested in <sequence>, <pick>, <flow>, <while>, <switch> elements.

Table 1 Mapping between WS-BPEL elements and process model constructs.

Structural construct	WS-BPEL element	Description
Task sequence	<sequence>	Allow for sequential execution of tasks.
A pair of *Or-Split/Join* gateways with conditions	<pick>	Perform the non-deterministic execution of one of several paths depending on an external event.
A *Loop* gateway with conditions	<while>	Perform a specific iterative task repeatedly until the given condition becomes false.
A pair of *Or-Split/Join* gateways with conditions	<switch>	Perform a conditional behaviour with a set of branches.
A pair of *And-Split/Join* gateways	<flow>	Perform parallel execution of a set of branches.
A synchronisation link	<link>	Support the synchronisation between tasks or gateways on the branches inside a <flow> element.
A scoped process fragment	<scope>	Originally used for defining the compensation scope for fault handling in WS-BPEL, yet here we use it to store the structural content and contextual information (such as variables and declarations), for scoped process fragments.
A dummy branch of a split/join structure	<empty>	Originally used to denote a dummy task, yet here we use it to stand for a dummy branch of a split/join structure.
Compensation handling process	<compensationHandler>	Specify the compensation process for a scoped process fragment.
Fault handling process	<faultHandlers>	Specify the fault handling process for a scoped process fragment.

5.2 Prototype Implementation

We have developed a prototype on the basis of SAP Maestro for integrating the process view abstraction and concretisation functions. This prototype is programmed in Java, and uses the packages from Tensegrity Software [25] for user interface design. Both original business processes and process views are written in WS-BPEL using XML syntax. The WS-BPEL process view transformation is enforced though XML transformations using the Extensible Stylesheet Language Transformation (XSLT) [30] by Apache Xalan [29]. Figure 9 illustrates the architecture and the view transformation procedure of the prototype. The slim arrows denote the behaviours between the functioning components, and the large white arrows denote the flow of input and output documents. Three repositories are used in the prototype. The *Process Hierarchy Repository* stores the process hierarchies for different business processes, and the *Mapping Repository* stores the mapping relations between process components and scoped process fragments, i.e., the content of functions γ and δ. Besides, the Mapping Repository also stores the hidden synchronisation links, as WS-BPEL documents do not support such invisible components.

This kernel system works as a backend system, whereby users can input the business processes/process views and abstraction/concretisation requests through diverse client systems, such as process editors, handset-based process viewers, web browsers, etc. When the system receives an abstraction/concretisation request and the inputted process view, the

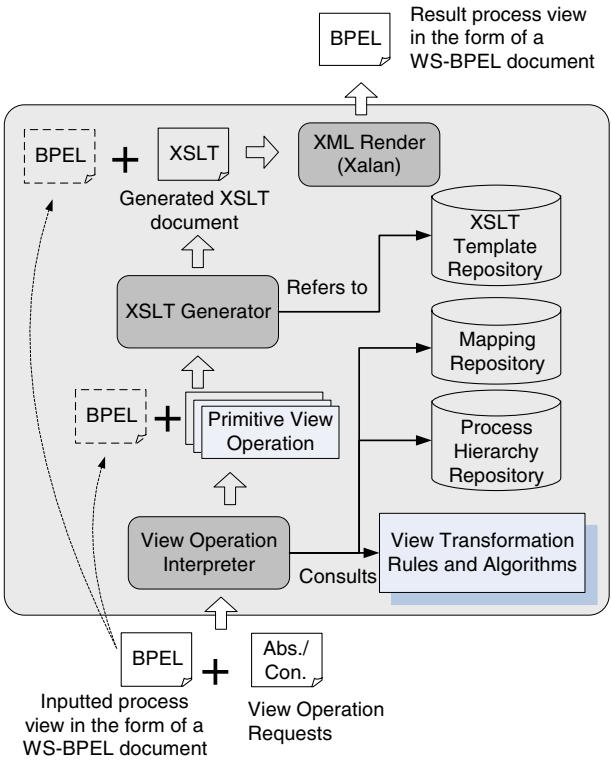

Figure 9 View generation system architecture.

View Operation Interpreter will decompose the requests into primitive process view operations, i.e., atomic task/link insertion and deletion operations, according to the developed algorithms. This decomposition will refer to the Process Hierarchy Repository and Mapping Repository. The derived primitive process view operations are passed to an *XSLT generator*, which will create an XSLT style file by filling a proper XSLT template from the XSLT Template Repository. The generated XSLT document is then sent to the *XML render* (Apache Xalan) to transform the original process view into an abstracted/concretised process view. The SAP Maestro is used to display the result process view graphically.

Now, we take the process view transformation from Figure 2(e) to Figure 2(c) as an example to illustrate how this system works. The view operation request from users is to concretise the task "Handle Shipping at DC" in Figure 2(e) into its mapped scoped process fragment at next level. According to the Mapping Repository and Process Hierarchy Repository, the View Operation Interpreter analyses and decomposes the view operation following the taskZoomIn algorithm. For this task concretisation operation, the View Operation Interpreter first checks the connection between the new scoped process fragment and the original process, combines process fragments, and adjusts the related synchronisation links, according to Rules 1, 2 and 3, as shown in the algorithm. Particularly, the combination of process fragments can be further converted into replacing the task with corresponding scoped process fragment, as indicated in Figure 10.

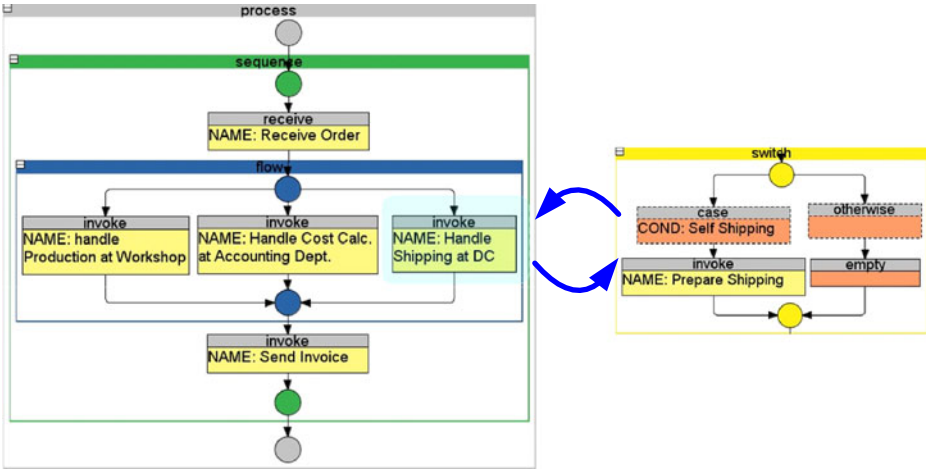

Figure 10 Replacing a task with a scoped process fragment.

Such replacement can be easily implemented with XSLT support. Figure 11 shows the XSLT style file created by the XSLT generator. This XSLT replaces task "handle shipping at distribution centre" with a <switch> element including a branch with task "prepare shipping" and a branch with an empty task.

The user interfaces of the FlexView-BPEL and the SAP Maestro are given in Figure 12.

<process> <!-- Declarations for partnerLinks, variables -> ... <sequence> <receive partnerLink="pLinkA" name="Receive PO"> ... </receive> ***<wait name="Handle shipping at DC "> ...*** ***</wait>*** <invoke partnerLink="pLinkB" name="Return Invoice"> ... </invoke> </sequence> </process>	This is the WS-BPEL document representing the process view shown in Figure 1 (e). Task "Receive Purchase Order". Task "Handle shipping at DC". Task "Return Invoice".
... <xsl:template match="/"> <xsl:apply-templates/> </xsl:template> <xsl:template match="*"> <xsl:choose> <xsl:when test="name(.)='***wait***' and @name='***Handle shipping at DC***'"> ***<scope> <switch> ... </switch></scope>*** </xsl:when> <xsl:otherwise> <xsl:element name="{name(.)}"> <xsl:for-each select="@*"> <xsl:attribute name="{name(.)}"> <xsl:value-of select="."/> </xsl:attribute> </xsl:for-each> <xsl:apply-templates/> </xsl:element> </xsl:otherwise> </xsl:choose> </xsl:template> ...	This is the generated XSLT style file for extending the view into the view shown in Figure 1 (c). Locate the specific task "Handle shipping at DC". Replace task "Handle shipping at DC" with the corresponding process fragment, i.e., the <scope> element that comprises the Or-split/join structure. Output the original content when task "Handle shipping at DC" is not located.

Figure 11 XSLT example.

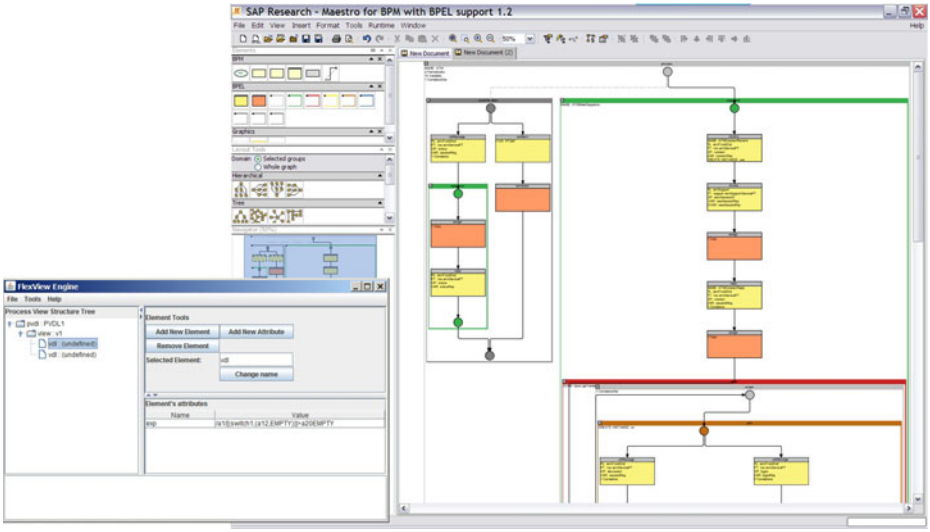

Figure 12 User interfaces of the FlexView-BPEL engine and the Maestro tool.

6 Related work and discussion

Business process technology has been an integral part of service oriented architecture (SOA) [14, 23]. Before WS-BPEL, HP's eFLow [6], Microsoft's BizTalk [18], Web service Flow Language (WSFL) [13], XLANG [26], etc., all intended to orchestrate services to fulfil complex business applications. Later, WS-BPEL [2] was developed to unify WSFL and XLANG, and became the widely accepted standard of business process language in the Web service environment. These languages well support sub processes which can be nested in a cascading hierarchy. At representation level, a sub process can be folded up into a special task, which can be extended back to the sub process. However, this fold-up function simply wraps up the sub process without concerning the structural elements across the sub process, e.g., the synchronisation links, fault handlers and compensation handlers that are also used outside of the sub process. In comparison, our approach can abstract a sub process into a task or edge through task aggregation and hiding operations. Particularly, the abstraction of a sub process into an edge indicates the hiding of process details, and is not supported by these modelling languages. Further, our approach takes into account the handling of cross-sub-process structural elements, including the synchronisation links, fault handles and compensation handlers, during the process view transformation.

Martens [17] discussed the verification on the structural consistency between a locally defined executable WS-BPEL process and a globally specified abstract process based on Petri net semantics. In regard to the structural consistency between general process views, Liu and Shen [15] proposed an order-preserving approach for deriving a structurally consistent process view from a base process. In their approach, the generation of "virtual activities" (compound tasks) needs to follow their proposed membership rule, atomicity rule, and order preservation rule. Recently, Eshuis and Grefen [9] formalised the operations of task aggregation and process customisation, and they also proposed a series of construction rules for validating the structural consistency. Compared with these work, first of all, our approach focused more on realising the process transformation at technical level

rather than theoretical level. Secondly, in the mentioned works, the customisation process actually lost some tasks. Yet, our approach preserved the hidden tasks and necessary mapping relations, and thus supported both abstraction and concretisation operations. Finally, synchronisation links were considered in our approach.

Some other works applied different workflow/process view approaches in the inter-organisational collaboration environment to support process privacy and interoperability. van der Aalst and Weske [28] proposed a "top-down" workflow modelling scheme in their public-to-private approach. Organisations first agree on a public workflow, and later each organisation refines the part it is involved in, and thereafter generates its private workflow. This work reflected a primitive idea of workflow view. Chiu et al. [8] borrowed the notion of 'view' from federated database systems, and employed a virtual workflow view for the inter-organisational collaboration instead of the real instance, to hide internal information. In [24], Schulz and Orlowska focused on the cross-organisational interactions, and proposed to deploy coalition workflows to compose private workflows and workflow views together to enable interoperability. Issam, Dustdar et al. [10] extracted an abstract workflow view to describe the choreography of a collaboration scenario and compose individual workflows into a collaborative business process. By deploying workflow views in the workflow interconnection and cooperation stages, their approach allows partial visibility of workflows and resources. Our previous works [31, 33] also established a relative workflow model for collaborative business process modelling. A relative workflow for an organisation comprises the local workflow processes of the organisation and the filtered workflow process views from its partner organisations. In this way, this approach can provide a relative collaboration context for each participating organisation. Some follow-up work targeted at the instance correspondence [34] and the process evolvement [32] in collaborative business processes, as well as role-based process view derivation and composition [35]. Supplementary to these works, our approach provided a practical implementation solution by incorporating the view concept into a popular standard business process modelling language. The abstraction and concretisation functions were naturally applicable to support privacy protection or perception control in the collaboration environment.

Proviado project [4] adopted process views for the personalised visualisation of large business processes. This project did some trade-off between the structural consistency and the adequate visualisation, whereby the generated process views were allowed not fully consistent with the original business process. Our work firmly complied with the proposed structural consistency and validity rules, and supported bi-directional process view operations.

Driven by the requirements for flexible business process representation, our FlexView framework shifts the process view concept to the technical level. The whole framework has been incorporated into WS-BPEL, and therefore can easily attach the process view support for Web service applications.

In summary, this work contributes to the following aspects:

1. Abstraction and concretisation functions towards process representations. With these two operations, users are allowed to choose and switch among different views of the same business process. In this way, our approach caters for the diversity of users' interests, authority levels, and so on. Although this paper chooses WS-BPEL as the target process modelling language, the conceptual and methodological part of our approach well supports other existing process modelling languages, such as Event-driven Process Chain (EPC) [27], Petri net based workflows, Business Process Modelling Notation (BPMN) [19], etc.

2. Information preservation and structural consistency mechanism. The defined rules and developed algorithms guarantee that the abstracted or concretised process views are consistent and valid in structure. Consequently, the two operations can be performed back and forth rather than one way only.
3. Incorporation of process views into WS-BPEL language. The whole framework is completely incorporated into WS-BPEL at both conceptual level and implementation level. A prototype is developed to provide an operational solution for supporting WS-BPEL process view abstraction and concretisation using XSLT techniques.

However, the integration of process view support also brings some trade-offs, which can be potential limitations. Some of the tradeoffs are summarised as follows, although they can be outweighed by the offered advantages.

- The process component model assumes that WS-BPEL processes are created with block structures. Therefore, non-block-structured business processes have to be converted into block-structured processes before applying our approach. The details about the conversion can be found in [12, 20].
- The diverse representation for the business process inevitably complicates the execution of the underlying business process. More coordination mechanisms are required for users and systems to run a specific business process with different process views.
- The process hierarchy needs to be predefined by process architects to enable the abstraction and concretisation of process representation.

7 Conclusions and future work

In this paper, we analysed the structural characteristics of WS-BPEL processes, and proposed a process component model to describe the structure and mapping relations of process fragments. Based on this process component model, a framework was developed to support the process view abstraction and concretisation functions for WS-BPEL processes. The process view transformation was under strict supervision of a set of rules on structural consistency and validity. The whole framework enabled to present a business process with different granularities to cater for the diverse requirements on process visualisation from users. A prototype was developed for the proof-of-concept purpose.

The future work included the strategies on automatic partitioning process fragments and generating the process hierarchy according to actual applications. With such support, the FlexView system is expected to assist business analysts, administrators to observe, diagnose business processes from different perspectives.

Acknowledgement The research work reported in this paper is supported by Australian Research Council and SAP Research under Linkage Grant LP0669660.

Appendix

The following content are extracted from [5].

Definition a1. *(Labelled Transition Systems)* A labelled transition system (LTS) is a tripule (*Proc*, *Act*, $\{\xrightarrow{a}|a\in Act\}$), where

– *Proc* is a set of states, ranged over by s;
– *Act* is a set of actions, ranged over by a;
– $\xrightarrow{a} \subseteq Proc \times Proc$ is a transition relation, for every *Act*. As usual, we shall use the more suggestive notation s in lieu of (s, s').

Definition a2. *(Observational Equivalence and Weak Bi-simulation)* A binary relation R over the set of an LTS is a weak bi-simulation if and only if for each action whenever s_1 R s_2 and a is an action:

– If $s_1 \xrightarrow{a} s_1'$, then there is a transition $s_2 \overset{a}{\underset{a}{\Rightarrow}} s_2$ such that $s_1' \approx s_2'$;
– If $s_2 \xrightarrow{a} s_2'$, then there is a transition $s_1 \Rightarrow s_1'$ such that $s_1' \approx s_2'$.

For two processes P and Q, $P \overset{\varepsilon}{\Rightarrow} Q$ if and only if there is a (possibly empty) sequence of τ transitions, i.e., unobservable transitions, that leads from P to Q. For each action a, we write $P \overset{a}{\Rightarrow} Q$ if and only if there are processes P' and Q' such that $P \overset{\varepsilon}{\Rightarrow} P' \xrightarrow{a} Q' \overset{\varepsilon}{\Rightarrow} Q$.

Milner's τ-laws:

$a.\tau.P \approx a.P$

$P + \tau.P \approx \tau.\mathrm{P}$

$a.(P + \tau.Q) \approx \mathrm{a}.(P + \tau.Q) + a.Q$

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web services—concepts, architectures and applications. Springer (2004)
2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services (BPEL4WS) 1.1. (2003)
3. Bertino, E., Squicciarini, A.C., Paloscia, I., Martino, L.: WS-AC: a fine grained access control system for web services. World Wide Web **9**(2), 143–171 (2006)
4. Bobrik, R., Reichert, M., Bauer, T.: View-based process visualization. In: The 5th International Conference on Business Process Management, Brisbane, Australia, pp. 88–95, 2007
5. Busi, N.: Process algebras, bisimulation (and logics), (2006)
6. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.-C.: Adaptive and dynamic service composition in eFlow. In: The 12th Conference on Advanced Information Systems Engineering, pp. 13–31, 2000
7. Charfi, A., Mezini, M.: AO4BPEL: an aspect-oriented extension to BPEL. World Wide Web **10**(3), 309–344 (2007)
8. Chiu, D.K.W., Karlapalem, K., Li, Q., Kafeza, E.: Workflow view based E-contracts in a cross-organizational E-services environment. Distributed and Parallel Databases **12**(2–3), 193–216 (2002)
9. Eshuis, R., Grefen, P.: Constructing customized process views. Data Knowl. Eng. **64**, 419–438 (2008)
10. Issam, C., Schahram, D., Samir, T.: The view-based approach to dynamic inter-organizational workflow cooperation. Data Knowl. Eng. **56**(2), 139–173 (2006)
11. Khoshafian, S.: Service oriented enterprise. Auerbach Publisher (2006)
12. Kopp, O., Martin, D., Wutke, D., Leymann, F.: On the choice between graph-based and block-structured business process modeling languages. In: MobIS 2008: Modellierung betrieblicher Informationssysteme, Stuttgart, Germany, pp. 59–72, 2008
13. Leymann, F.: Web Services Flow Language (WSFL) 1.0, (2001)
14. Leymann, F., Roller, D., Schmidt, M.-T.: Web services and business process management. IBM Syst. J. **41**(2), 198–211 (2002)
15. Liu, D.-R., Shen, M.: Workflow modeling for virtual processes: an order-preserving process-view approach. Inf. Syst. **28**(6), 505–532 (2003)
16. Liu, C., Li, Q., Zhao, X.: Challenges and opportunities in collaborative business process management. Information System Frontiers (2008)

17. Martens, A.: Consistency between executable and abstract processes. In: The 7th IEEE International Conference on e-Technology, e-Commerce, and e-Services, Hong Kong, China, pp. 60–67, 2005
18. Microsoft BizTalk (http://www.microsoft.com/biztalk/)
19. OMG: Business process modeling notation (BPMN 1.1), (2008)
20. Ouyang, C., Dumas, M., ter Hofstede, A., van der Aalst, W.M.P.: Pattern-based translation of BPMN process models to BPEL web services. Int. J. Web Serv. Res. **5**(1), 42–62 (2008)
21. Papazoglou, M.P.: Web services and business transactions. World Wide Web **6**(1), 49–91 (2003)
22. Papazoglou, M.: Web services: principles and technology. Prentice Hall (2007)
23. Papazoglou, M.P., Yang, J.: Design methodology for web services and business processes. In: The 3rd International Workshop on Technologies for E-Services, pp. 54–64, 2002
24. Schulz, K.A., Orlowska, M.E.: Facilitating cross-organisational workflows with a workflow view approach. Data Knowl. Eng. **51**(1), 109–147 (2004)
25. Tensegrity Software (http://www.tensegrity-software.com/home/home.html)
26. Thatte, S.: XLANG—web services for business process design, (2001)
27. van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Inf. Softw. Technol. **41**(10), 639–650 (1999)
28. van der Aalst, W.M.P., Weske, M.: The P2P approach to interorganizational workflows. In: International Conference on Advanced Information Systems Engineering, pp. 140–156, 2001
29. Xalan (http://xml.apache.org/xalan-j/)
30. XSLT (http://www.w3.org/TR/xslt)
31. Zhao, X., Liu, C.: Tracking over collaborative business processes. In: The 4th International Conference on Business Process Management, pp. 33–48, 2006
32. Zhao, X., Liu, C.: Version management in the business process change context. In: The 5th International Conference on Business Process Management, Brisbane, Australia, pp. 198–213, 2007
33. Zhao, X., Liu, C., Yang, Y.: An organisational perspective on collaborative business processes. In: The 3rd International Conference on Business Process Management, Nancy, France, pp. 17–31, 2005
34. Zhao, X., Liu, C., Yang, Y., Sadiq, W.: Handling instance correspondence in inter-organisational workflows. In: The 19th International Conference on Advanced Information Systems Engineering, Trondheim, Norway, pp. 51–65, 2007
35. Zhao, X., Liu, C., Sadiq, W., Kowalkiewicz, M.: Process view derivation and composition in a dynamic collaboration environment. In: The 16th International Conference on Cooperative Information Systems, Monterrey, Mexico, pp. 82–99, 2008
36. Zhao, X., Liu, C., Sadiq, W., Kowalkiewicz, M., Yongchareon, S.: WS-BPEL business process abstraction and concretisation. In: The 14th International Conference on Database Systems for Advanced Applications, Brisbane, Australia, 2009