An approach for business process model registration based on ISO/IEC 19763-5

Zaiwen Feng, Chen Wang, Chong Wang, Yi Zhao, Dickson K.W. Chiu, Keqing He

Abstract—To facilitate business collaboration and interoperation among enterprises, it is critical to discover and reuse appropriate business processes modeled in different languages and stored in different repositories. However, the formats of business process models are very different, which makes it a challenge to fuse them in a unified way without changing their original representations and semantics. To solve this problem, this paper uses semantic interoperability technique, which is able to transform heterogonous process models into uniform registered items. Based on the general and unambiguous metamodel for process model registration (PMR for short) that we proposed before, in this article, we provide a generic process model registration framework for registering heterogeneous business process models to facilitate semantic discovery of business processes across enterprises, and promote process interoperation and business collaboration. Considering Event-driven Process Chain (EPC) is a popular process model widely used in the industry, we focus on the mapping rules and related algorithms from EPC to PMR and develop an automatic process model registration tool for EPC. Moreover, we conduct a series of experiments to verify the correctness and efficiency of our proposed framework by leveraging the real data set of 604 EPCs from SAP.

Index Terms—business process model; EPC; process repository, semantic interoperation

1 INTRODUCTION

Rapid progress of economic globalization brings greater and more frequent collaboration between businesses around the globe, demanding solutions for a wide range of ever complicated interoperability problems [1]. To share knowledge, the discovery and reuse of business process is an effective way to improve the interoperability of various existing business processes in different enterprises. Process designers may reuse similar existing process models via discovering them in the repository to improve efficiency and correctness of process modeling. However, business process models are currently designed with various kinds of modeling language and dispersed in different repositories, with different motivations and approaches. Although different modeling languages might share some similar concepts, the differences between semantics and grammar are distinct. Therefore, there are no trivial one-to-one mapping relations among these models built with different modeling languages. Such problems hinder the reuse of cross-domain and cross-enterprise business processes, causing great barriers in deep collaboration among enterprises.

Aiming to address these problems, our initial work [5,25,26,30] advocated the use of model-driven and ontology-supported methodologies to facilitate process interoperation for enterprise collaboration by providing a common and unambiguous metamodel. A process normal form named PMR metamodel captures the essential information of heterogeneous business processes in a unified way. By leveraging this normal form, we can standardize the registration of business process models and further facilitate business process reuse, integration, and collaboration. However, our previous work has not yet provided a general and logical-level framework that is able to guide the registration of heterogeneous business process. Besides that, the prototype needs to be improved to support more process languages.

The objective of this article is to enable registration for heterogeneous business process models based on ISO/IEC 19763-5 to facilitate the semantic discovery of business processes across enterprises. Existing technologies has enabled some aspects of addressing this issue. To date, a large number of languages have been proposed for specifying and executing transformations between models conforming to different metamodels. The ability to automatically transform between models expressed in different languages (metamodels) is of importance to the wide-spread adoption of Model-Driven Development (MDD). Although various approaches to automated model transformation have been proposed, the current consensus is that specialized languages, such as QVT [33], ATL [6] and ETL [34], which provide a mixture of declarative and imperative constructs, are most suitable for specifying model transformations. Model-driven methods have also been exploited to standardize process modeling and

support interoperation between these models [10]. There are also some other transformation approaches particularly used in a pair of specified process modeling notation, like [9,20]. Although general transformation languages like ATL [6], QVT [33] or programming languages like Java [12] provide very comprehensive model transformation, there is no transformation framework particularly aiming at business process model transformation.

The purpose of PMR is to allow analysts to conduct federated query across language-specific process model repositories. So, the PMR metamodel does not keep all of the metadata from a language-specific process model. The exposed registration information can be abstract and incomplete. However, it has to be sufficient to discover reusable business processes and locate them in the repositories. Contemporary business process query languages, like APQL [7], support not only structural process characteristics but also process model *behavior*. Accordingly, the behavior semantics of language-specific process models must be preserved after transformation to PMR graphs. In other words, every behavior captured in the language-specific process model should also be captured by the PMR graph. This requirement ensures that the execution semantics of the language-specific process model is not lost in the PMR graph.

With the aim of registering language-specific process models into PMR, this article proposes a general framework and algorithm of mapping heterogonous business process model to PMR metamodel. A language-specific process model can be mapped to PMR while preserving its behavioral semantics. Specially, considering Event-driven Process Chain (EPC) is a popular process model widely used in the industry, we specify the mapping rules and related algorithm from EPC to PMR metamodel as an instantiation of our approach. The main contributions of this work are threefold:

1. We propose a generic registration framework that maps a process model in various process languages to PMR registration items. Taking EPC as an example, we illustrate how to specify mapping rules and algorithms from a specific process to PMR metamodel.
2. We conduct experiments to evaluate the correctness and the performance of our mapping algorithm.
3. We have implemented an open-source tool to facilitate PMR based on our framework.

The rest of the article is organized as follows: Section 2 discusses related work. Section 3 describes our previous work on PMR metamodel. Section 4 presents the framework of mapping diverse business process modeling languages to PMR, and the algorithm design on the mapping from EPC to PMR. Section 5 details some experiments conducted to demonstrate the effectiveness and efficiency of our approach. Section 6 concludes this paper with our future work directions.

2 Related Work

Since the work in this article focuses on how to register heterogonous business process models into the common manageable atoms based on PMR metamodel, we put the emphasis on the mapping or conversion approaches of heterogonous business process models.

Currently, as far as we know, there is no model conversion approach particularly for business process transformation. However, there exist a lot of Model-to-Model (M2M) transformation approaches and languages for the task of transforming general models, like QVT [33], ATL [6] or ETL [34].

QVT is the current OMG standard for model transformation. QVT adopts a hybrid style by providing both declarative and imperative constructs. With regard to integration, the OMG has also standardized a model-to-text transformation language (MOF2Text) [8] that reuses parts of QVT. The MOF model-to-text standard addresses how to translate a model to various text artifacts such as code, deployment specifications, reports, documents, etc. A template-based approach is used wherein a *Template* specifies a text template with placeholders for data to be extracted from models. These placeholders are essentially expressions specified over metamodel entities with queries being the primary mechanisms for selecting and extracting the values from models. These values are then converted into text fragments using an expression language augmented with a string manipulation library.

ATL [6] (ATLAS Transformation Language) is a domain-specific language for specifying model-to-model transformations. The general transformation pattern of ATL is that a source model is transformed into a target model according to a transformation rule written in the ATL language. Transformation rule is the basic construct in ATL used to express the transformation logic. ATL rules may be specified either in a declarative style or in an imperative style. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models. A model transformation case study

from Web Service Choreography Description Language (WSCDL) [32] to BPEL is presented that uses ATL as transformation languages [31].

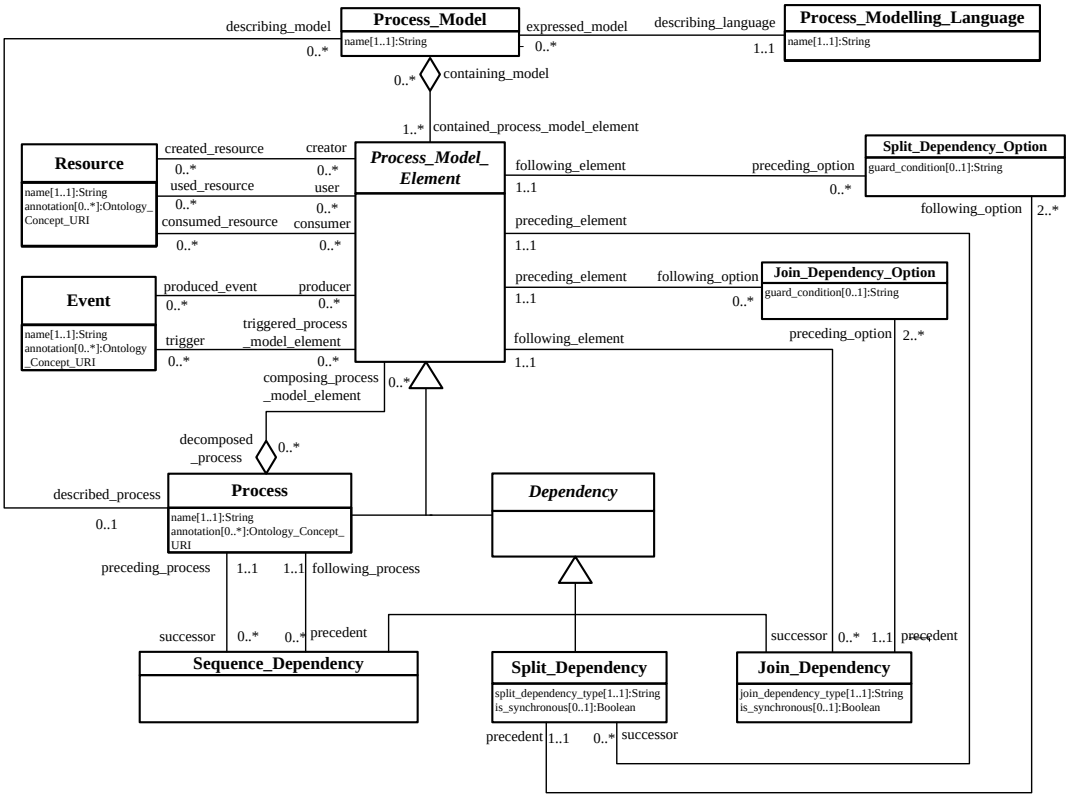
ETL [34] (Epsilon Transformation Language) is a hybrid model transformation language that is integrated with a number of additional purposes such as model comparison, merging, validation, and model-to-text transformation to help realize complex model management workflows. Unlike most contemporary model transformation languages, ETL is capable of transforming an arbitrary number of source models into an arbitrary number of target models. ETL adopts a hybrid style and features declarative rule specification using advanced concepts such as *guards*, *abstract*, *lazy,* and *primary* rules, and automatic resolution of target elements from their source counterparts.

SiTra [12] (Simple Transformer, SiTra) is a minimal, Java based library that can be used to support the implementation of many practical transformations. The underlying idea of SiTra is to put less focus on the specification language, maintenance, and documentation aspects of transformation, by focusing on the implementation of transformations. SiTra uses Java for the specification of transformations. This relinquishes the programmer from learning a new language for the specification transformation. A case study is reported in [18] to illustrate transformations from Ontology Web Language-Service (OWL-S) [2] to Business Process Execution Language (BPEL) [13].

However, all these model transformation frameworks are not tailored to the specific requirements for business process model transformation.

There exist a bundle of works that convert one specific process language to another, for example, from EPC to Petri Net [20], from BPMN to BPEL [9], from BPEL to EPC [23], and so on. This sort of transformation aims at *preserving the semantics* from a model notation to another. However, this assumption does not work for our approach because PMR metamodel only captures the *minimal common metadata* about processes, aiming to facilitate the federated discovery and re-use of process models expressed in diverse languages and stored in different repositories.

There are also some frameworks that map heterogonous process notations to a normal and neutral metamodel. In these work, different kinds of process metamodels have been proposed in order to satisfy a specific requirement. For example, IPM Executable Process Definition Language [19] is designed to support flexible process specification of organizational structures and role models. Process information can be exchanged with the repository through the IPM EPDL, which is an XML-based format that can be used to store information about process models and activities, control flow, organizational structure, authorization and resource assignment, data and monitoring. Marcello *et al*. [10] built an Advanced Process Model Repository (APROMORE), which aims to hold, analyze and re-use large sets of process models. The re-use of process models in APROMORE is based on a *canonical process format*, which provides a common, unambiguous representation of business processes captured in different notations and at different abstraction levels. Oryx [15] is a Web-based process modeling tool that supports users browsing, creating, storing, and updating process models online. The tool uses a repository for storing the business process models that are created with it. Oryx mainly focuses on the activity and control-flow aspect. It supports many process notations, including BPMN, EPC, Petri nets, FMC Diagram, and XForms. However, as far as we know, no conversion framework and algorithms have been given in these works.



NOTE Metaclasses whose names are italicized are abstract metaclasses

Fig. 1. The metamodel of PMR (adapted from [5]).

3 PRELIMINARIES

In the past decade, a couple of business process model repositories have been developed, which in general is employed to store business process models described in a specified process language, for example, IBM BPEL (Business Process Execution Language) Repository [21], BPMN (Business Process Model and Notation) Repository [22]. In order to promote business process collaboration among enterprises, we provided a metamodel for the registration and indexing of the existing process models in our previous work [5,25,26,30]. PMR is mainly designed to register business process automatically, facilitate the semantic discovery of business processes across enterprises, and promote process interoperation and business collaboration. For this purpose, it only provides *selected* metadata and *common* semantics of process models created with a specific process modeling language, including BPMN [11], BPEL [13], EPC [24], OWL-S Process [2], UML[27], PSL[28] and IDEF3[29]. The metamodel can help discovery of the function and composition of a process, and promote reuse of its components at different levels of granularity. Fig. 1 shows an overview of our PMR metamodel.

For PMR, a process model is used as a representation of a process, and it describes the contained process model elements using a specified process modeling language. The process model elements include processes and dependencies among processes and other process model elements. For each process model element, there are some events that can be used to trigger a process model element or to be produced by a process model element.

Dependencies represent the control constraints among processes represented by a process model. A dependency can be specialized as a sequence dependency, a split dependency, or a join dependency. A sequence dependency specifies that the processes are executed in order. A split dependency specifies that when the preceding process model element is completed, one or more of the following process model elements will be executed in parallel. A join dependency specifies that the following process

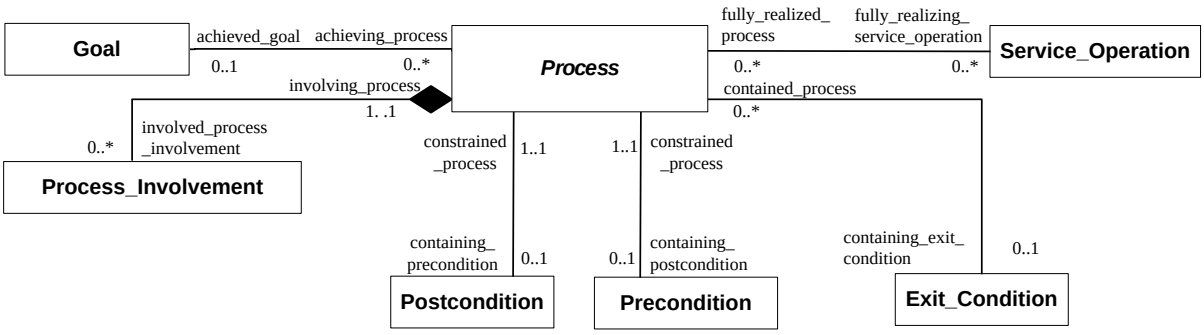


Fig. 2. The metamodel of PMR: Part 2[5].

model element will start when the selected preceding process model elements are completed. A split dependency type is used to specify a logical gate for the following processes. Similarly, a join dependency type is used to specify a logical gate for the preceding processes. The logic of both a split dependency type and a join dependency type can be XOR, OR, and AND. For a split dependency type, XOR means that one and only one of the succeeding process model elements is allowed to execute, OR means that one or more of the succeeding process model elements are allowed to execute, and AND means that all of the succeeding process model elements must be executed. For a join dependency type, XOR means that the succeeding process model element executes if one and only one of the preceding process model elements completes successfully, OR means that the succeeding process model element executes if one or more of the preceding process model elements completes successfully, and AND means that the succeeding process model element executes if, and only if, all of preceding process model elements completes successfully. In addition, a split dependency option represents the guard conditions of the following process model elements to be executed after the value of a split dependency type is decided. Similarly, a join dependency option specifies the guard conditions of the preceding process model elements to be executed after the value of a join dependency type is decided.

Particularly, processes, resources, and events can be annotated with zero, one or more concepts of domain-specific ontologies, which expose essential information of heterogeneous business processes in a unified way to promote business process collaboration and interoperation.

Fig. 2 shows the associations between process models, business goals, and services, which are important task resources for process model. The association between process and goal specifies that each process achieves zero, one, or more goals, and each goal is achieved by zero, one, or more processes. A goal may exist that is not specified to be achieved by a process, and a process may exist, which is not applied to achieve a specific goal. Similarly, each process involves zero, one, or more process involvements, where each process involvement denotes the involvement of a role with a process, such as actor or beneficiary. Each process involvement indicates that a role is involved in the execution of one and only one process. A process involvement shall have exactly one associated process.

The association between process model and service model specifies that each process is fully realized by zero, one, or more service operations, and each service operation can fully realize zero, one, or more processes. A process may exist that is not specified to be realized by a service, and a service may exist that is not applied to realize a process. Each process may have one pre-condition and/or one post-condition. A process may exist with no associated pre-condition or post-condition. Each process has zero or one exit condition to state a set of conditions that will exist to cause a process to terminate before its completion.

4 OUR APPROACH

Our preliminary work described the details of the PMR and its elements. However, it is still a problem that how different language constructs are aligned together in the process model registration, how language constructs exist in the original languages, and how they will be in PMR. These problems further boil down to firstly, how to *decompose* the existing business process model described by a specified process languages into language-specific manageable atoms, and secondly, how to *reconstruct* these language-specific manageable atoms onto PMR graph. In this section, we generalize our registration method and point out how these process models in the original languages are mapped into

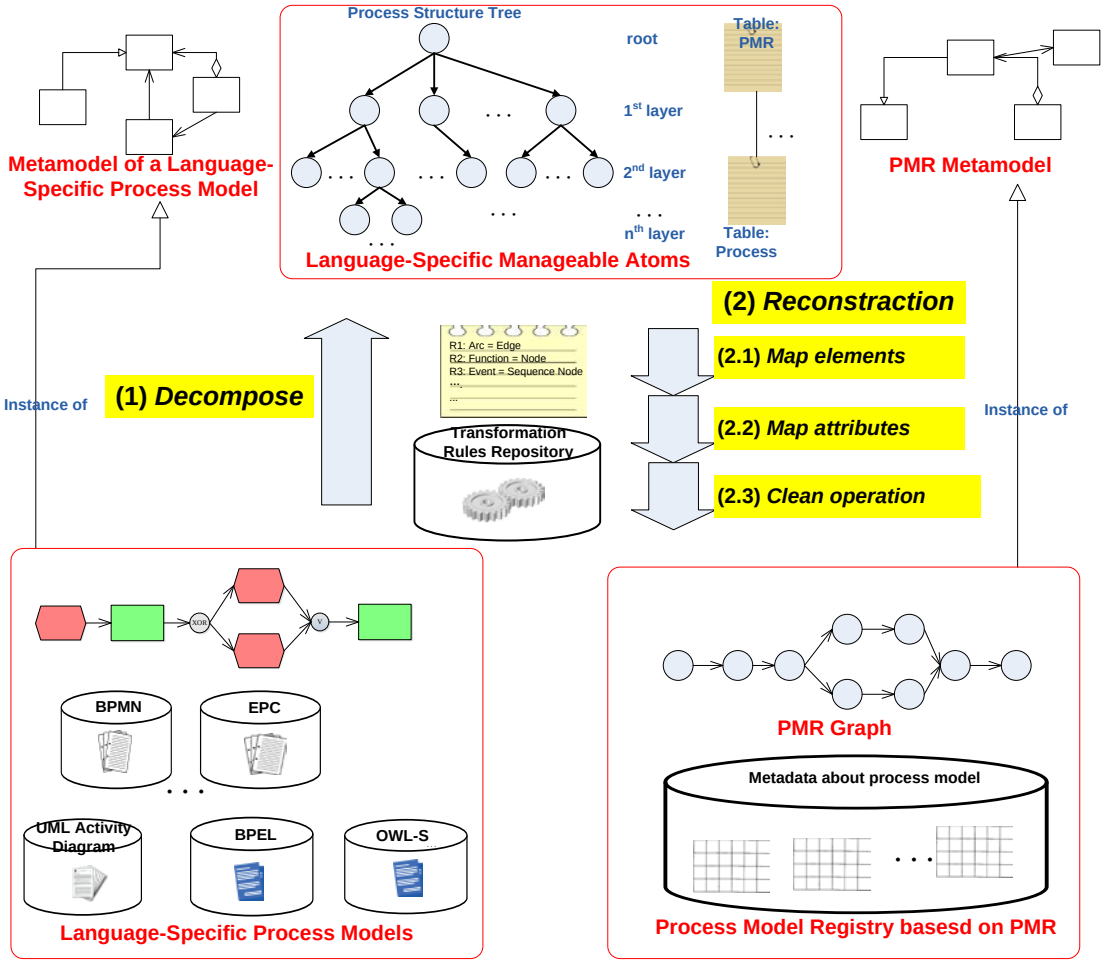


Fig. 3. Registration Framework based on PMR.

PMR in a uniform way.

The *process registration framework* based on PMR is shown in Figure 3. The objective of the process registration framework is to provide a generic solution of integrating heterogeneous business process models based on the selected metadata provided by the PMR metamodel.

First, a language-specific process model is decomposed based on *process structure trees* (PSTs). A PST represents the abstract hierarchical structure of a process model. Each node of a PST represents the model elements of a process while it records dependency information such as predecessor(s) and successor(s) and its parent node as well. Each edge of a PST denotes a sub-process relationship. A PST is designed to be implemented on top of standard relational databases, which is called *language-specific manageable atoms*.

Next, language-specific manageable atoms are mapped to PMR-based manageable atoms based on language-specific *mapping rules*, which correspond to elements and attributes pertained to a specific process language to the elements or attributes in the PMR model. The reconstruction step consists of three sub-steps, i.e. mapping elements, mapping attributes and cleaning redundant nodes inside PMR graphs. Then, PMR-based manageable atoms are ingested into the process register eventually.

To implement registration of business processes in different languages, it is necessary to define the mappings from the specific process modeling language to the PMR metamodel. Due to the popularity of application of EPC in industry domain, we take EPC as an example in this article to illustrate how to define the mapping rules and the corresponding algorithms for implementing process registration.

4.1 Model Decomposition

In the phase of decomposing an existing process model, a language-specific process model, such as

EPML (for EPC), BPEL, or OWL-S Process model, is parsed and decomposed into language-specific manageable atoms. No matter *block-oriented* languages (such as BPEL and OWL-S Process Model) or *graph-oriented* languages (such as EPCs, BPMN, and UML Activity Diagram), the decomposing phase is implemented by the following two steps.

- **Step1:** Decompose the original process model into a *process structure tree*.
- **Step2:** Build the common language-specific manageable atoms.

A language-specific process model is decomposed based on *process structure tree (PST)*. A PST describes the abstract hierarchical structure of a language-specific process model. Besides, it records the dependency information, such as *parent*, *predecessor*, and *successor*, for each process model element. The root of PST represents the whole process model. The nodes of a PST represent the model elements of a process model.

4.1.1 A Tree Representation of a Language-Specific Process Model

A language-specific process model is represented as a PST. The nodes of a PST contain a process model's common structural manageable atoms. Each node has a name, which is also the name of the process model element that is represented by the node. PSTs are designed to represent hierarchical structures of any kind of process model. Each edge of PST denotes a *sub-process relationship*.

For a *graph-based* process model (e.g. EPC), each *leaf node* of PST represents a graphical notation of a process model, for example, XOR connector, function or event in EPC model. At the same time, each *non-leaf* node also represents the root of a sub-process. The number of non-leaf layers of PST depends on the maximum nested layers of the process model. For instance, the number of nested layers of EPC model in Fig. 4a is 2. Accordingly, the number of non-leaf layers of its PST is also 2. For a *block-oriented* process model (e.g., OWL-S, BPEL), each node of PST represents a structural activity of process model. For example, an OWL-S process model contains tags that represent the underlying document structure, e.g., $process{:}Composite{:}Process$, $list{:}first$, $list{:}rest$, which can be exploited to detect the structure. The nesting of structured activities is preserved as functions with sub-process relationships in PST. The number of non-leaf layers of PST depends on not only the maximum nested layers of structural process definition, but also the maximum nested layers of process model.

The PSTs we consider are unordered trees. That is, the children of a node in a PST do not have a fixed order. However, *the dependency information* between nodes of a process model is attached to the nodes of a PST. PST is composed of three types of nodes, i.e. *root nodes*, *non-leaf nodes*, and *leaf nodes*,. The functionalities and index structures of these three types of nodes are illustrated as follows.

- Root node:

The root of a PST represents the root of the process model. A standard relational table is designed for the PST root node, i.e., $Process(ProcessId, Languge, PMEId, ProcessType)$. Table *Process* contains all the language-specific process models. $ProcessId$ is automatically generated by incrementing a counter every time when a new language-specific process model is registered. $Language$ is the process notation that a new language process model to be registered. For instance, a process model in the format of EPC is demonstrated in Fig. 4a. The $ProcessId$ of this process is "1" and the $Language$ of this process is "EPC". $PMEId$ consists of the entire modeling element units decomposed from the original process model. For example, the process model in Figure 4a contains 10 process model elements. Each of the process model element has a unique id, i.e. 01, 02,..., 10, respectively. Each modeling element corresponds to a node in PST. $ProcessType$ represents the type of process model to be registered, i.e., graph-based or block-oriented. For instance, the process model to be registered in Fig. 4 is graph-based as it is an EPC model.

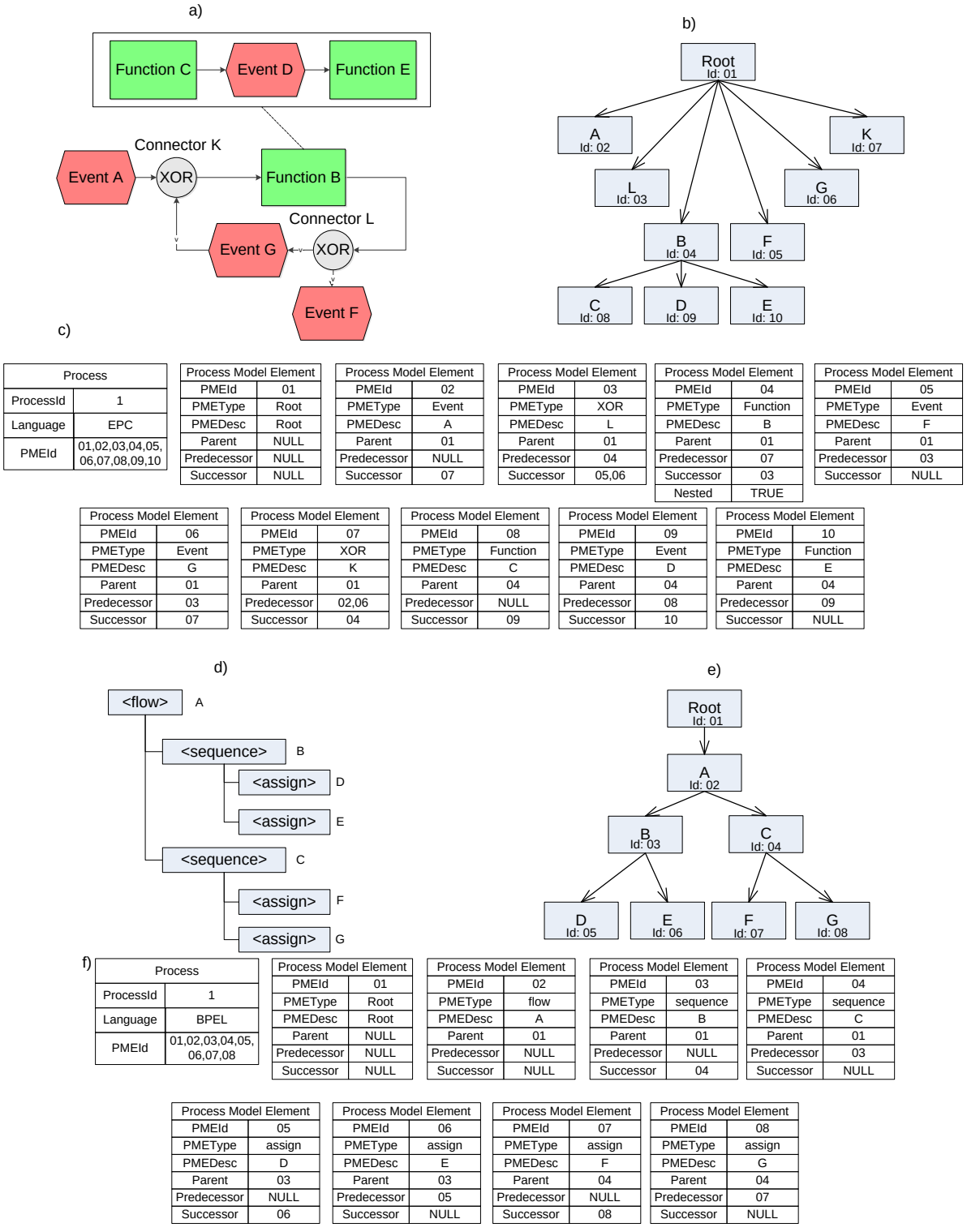
- Non-leaf node

The non-leaf node of a PST represents a process model element to be transformed. For a graph-based process model, a non-leaf node of PST represents a graphical notation of process model, for example, XOR connector, function or event in EPC model. For a block-oriented process model, a non-leaf node of PST means a structural element of the process model, for example, a structural activity $< pick >< /pick >$ in BPEL process can be corresponding to a PST node.

Table *PME* is the main component of the language-specific manageable atoms. A standard relational table is designed for the PST non-leaf nodes, i.e., $PME(PMEId, PMEType, PMEDecs, Parent, Predecessor, Successor)$. $PMEId$ assigns a unique identifier to each indexed PME, where $PMEDesc$ is the text label of a PME, $PMEType$ is the type of this PME, $Parent$ contains the identifier of the superior process model element of this PME, $Predecessor$ and $Successor$ refer to the identifiers of the predecessor and successor of this PME, respectively.

- Leaf node

A non-leaf node of a PST also represents a process model element to be transformed. Different from non-leaf nodes, the process model element that a leaf node corresponds to cannot be nested. We also use a relational table to manage PST leaf nodes, i.e., $PME(PMEId, PMEType, PMEDecs, Parent, Predecessor, Successor)$.



Process	
ProcessId	1
Language	EPC
PMEId	01,02,03,04,05, 06,07,08,09,10

Process Model Element	
PMEId	01
PMEType	Root
PMEDesc	Root
Parent	NULL
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	02
PMEType	Event
PMEDesc	A
Parent	01
Predecessor	NULL
Successor	07

Process Model Element	
PMEId	03
PMEType	XOR
PMEDesc	L
Parent	01
Predecessor	04
Successor	05,06

Process Model Element	
PMEId	04
PMEType	Function
PMEDesc	B
Parent	01
Predecessor	07
Successor	03
Nested	TRUE

Process Model Element	
PMEId	05
PMEType	Event
PMEDesc	F
Parent	01
Predecessor	03
Successor	NULL

Process Model Element	
PMEId	06
PMEType	Event
PMEDesc	G
Parent	01
Predecessor	03
Successor	07

Process Model Element	
PMEId	07
PMEType	XOR
PMEDesc	K
Parent	01
Predecessor	02,06
Successor	04

Process Model Element	
PMEId	08
PMEType	Function
PMEDesc	C
Parent	04
Predecessor	NULL
Successor	09

Process Model Element	
PMEId	09
PMEType	Event
PMEDesc	D
Parent	04
Predecessor	08
Successor	10

Process Model Element	
PMEId	10
PMEType	Function
PMEDesc	E
Parent	04
Predecessor	09
Successor	NULL

Process	
ProcessId	1
Language	BPEL
PMEId	01,02,03,04,05, 06,07,08

Process Model Element	
PMEId	01
PMEType	Root
PMEDesc	Root
Parent	NULL
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	02
PMEType	flow
PMEDesc	A
Parent	01
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	03
PMEType	sequence
PMEDesc	B
Parent	01
Predecessor	NULL
Successor	04

Process Model Element	
PMEId	04
PMEType	sequence
PMEDesc	C
Parent	01
Predecessor	03
Successor	NULL

Process Model Element	
PMEId	05
PMEType	assign
PMEDesc	D
Parent	03
Predecessor	NULL
Successor	06

Process Model Element	
PMEId	06
PMEType	assign
PMEDesc	E
Parent	03
Predecessor	05
Successor	NULL

Process Model Element	
PMEId	07
PMEType	assign
PMEDesc	F
Parent	04
Predecessor	NULL
Successor	08

Process Model Element	
PMEId	08
PMEType	assign
PMEDesc	G
Parent	04
Predecessor	07
Successor	NULL

Fig. 4. Example of a) EPC process model, b) PST of the EPC model, c) the common language-specific manageable atoms of the EPC model, d) BPEL process model, e) PST of BPEL model, and f) the common language-specific manageable atom of BPEL model.

4.2 Model Reconstruction

The model reconstruction phase maps manageable atoms in a language-specific process model to PMR-based manageable atoms. The common targeting format for mapping is *PMR graph*, which is the instance of PMR metamodel described in our preliminary work. The definition of PMR graph is shown as follows.

Definition 1: PMR graph $G_{PMR} = (N, Pre, Eff, T, Typ, L, E, \rho, \pi, \varepsilon, \lambda, \gamma, \tau)$ consists of six finite sets N, Pre, Eff, T, Typ, and L. Furthermore, the node set N is the disjoint union of its subsets, i.e., $N = N_p \cup N_r \cup N_s \cup N_{sd} \cup N_{jd} \cup N_{sdo} \cup N_{jdo} \cup N_e$, a binary relation $E \subseteq N \times N$, a surjective function $\rho: N_p \to Pre$, a surjective function $\pi: N_p \to eff$, a surjective function $\varepsilon: (N_s \cup N_{sd} \cup N_{jd}) \to T$, a partial function $\lambda: (N \cup E) \mapsto L$, a function $\gamma: N_{sd} \to Typ$, and a function $\tau: N_{sj} \to Typ$ such that:

- We write $N = N_p \cup N_r \cup N_s \cup N_{sd} \cup N_{jd} \cup N_{sdo} \cup N_{jdo} \cup N_e$ for all nodes of the PMR graph.
- N_p is a finite set of the *process* nodes.
- N_r is a finite set of *resource* nodes.
- N_s is a finite set of *sequence* nodes.
- N_{sd} is a finite set of *split dependency* nodes.
- N_{jd} is a finite set of *join dependency* nodes.
- N_{sdo} is a finite set of *split dependency option* nodes.
- N_{jdo} is a finite set of *join dependency option* nodes.
- N_e is a finite set of *event* nodes.
- Pre is a finite set of pre-conditions for an process node. Each precondition represents a condition that should be true before an activity node is executed.
- Eff is a finite set of effects for an process node. Each effect represents a condition that should be true at the completion of execution for an activity node.
- T is a finite set of transitional expression, each of which is used as a property to save the event text label temporarily by the split and join dependency nodes.
- Typ is a finite set of logical types for split dependency nodes N_{sd} and join dependency nodes N_{jd}, such as AND, OR, and XOR.
- E is a set of sequence flows. Each sequence flow $e \in E$ represents a directed edge between two nodes.
- We write $U = N_p \cup N_r \cup E$ for all *units* of the PMR graph which can carry a label.
- L is a finite set of text labels.
- The surjective function ρ specifies the assignment of a set of pre-conditions Pre to process nodes N_p. Hence, every process node can be associated with one or more pre-conditions.
- The surjective function π specifies the assignment of a set of effects eff to process nodes N_p. Hence, every process node can be associated with one or more effects.
- The surjective function ε specifies the assignment of a piece of transitional expression $t \in T$ to sequence, split and join dependency nodes $N_s \cup N_{sd} \cup N_{jd}$. Hence, every split and join dependency node can be associated with a piece of transitional expression.
- The partial function λ defines the assignment of a label $l \in L$ to a process model unit $u \in U$;
- The function γ specifies the type of a split dependency $n_{sd} \in N_{sd}$ as *AND, OR, XOR*.
- The function τ specifies the type of a join dependency $n_{sj} \in N_{sj}$ as *AND, OR, XOR*.

The reconstruction of manageable atoms is further reduced into three sub-steps, which are illustrated below:

- **Step1**: We create an intermediate graph based on the common manageable atoms of language-specific business process model. The creation of an intermediate graph is according to the pre-defined mapping rules, which map manageable atoms of language-specific business process model to PMR graph elements.
- **Step 2:** We map element attributes of a language-specific process model to the corresponding

TABLE 1. MAPPING RULES FOR CORE ELEMENTS

Rule No.	*EPC Element*	*PMR Graph Element*
R1	Arc	Edge
R2	Function	Process Node
R3	Connector (in-degree > 1 & out-degree = 1)	Join Option Node connected to Join Dependency Node
R4	Connector (in-degree=1 & out-degree > 1)	Split Dependency Node connected to Split Option Node
R5	Event (Predecessor: function & Successor: function)	Sequence Node
R6	Event (in-degree = 0 | out-degree = 0)	Event Node
R7	Event (Predecessor : Connector & out-degree!= 0)	The Event is removed

element attributes of PMR graph based on pre-defined mapping rules. Part of attribute value is propagated and reassigned inside PMR graph.

- **Step 3:** Some redundant nodes are removed to simply the PMR graph.

The reconstruction of manageable atoms is based on a series of mapping rules, which defines the correspondence of elements and attributes between a specific process language and the PMR model. It is necessary to define the mapping rules to PMR customized for each process language. In our previous work, we have already defined mapping rules from BPEL to PMR [25,26] and from OWL-S to PMR [30].

In the rest of this section, we first present the meta-rule for mapping elements to PMR, followed by the concrete rules from EPC to PMR as an example. Next, the meta-rule for mapping attributes to PMR is also presented with rules of mapping attributes from EPC to PMR as an instance. Then, a general transformation algorithm is proposed as the core of our framework. Next, we detail a solution considering a common scenario that text label of event is propagated in the chain of dependency nodes in an intermediate PMR graph. Lastly, we introduce a reduction rule to remove the unnecessary elements and obtain the final PMR graph.

4.2.1 Mapping Elements

As one of the generic phases, all core elements of a specified process model will be first mapped to the corresponding elements in the PMR metamodel one by one, and the skeleton of PMR graph is created at the same time.

The meta-rule for mapping elements from a language-specific process model to PMR graph is shown below.

$$Element(X) \wedge Predecesor(Y) \wedge Successor(Z) \wedge Parent(W) \wedge Constraint(c_1, c_2, .., c_m) \Rightarrow ElementInPMR(V)$$

This rule describes that a process model element X in a language-specific process model is mapped to an element V in PMR graph under the condition that Y is the predecessor of X, Z is the successor of X, and W is the parent of X, and the constraints $c_1, c_2, .., c_m$ on X are satisfied. Here, Y, Z, and W denote the process elements in a language-specific process model.

Let us take the conversion from an EPC to PMR graph as an example. EPC core elements comprise *Function*, *Connector*, *Arc,* and *Event*. All the core elements will be converted first during the mapping procedure. The mapping rules for core elements are shown in Table 1.

1) *Arc* in EPC is used to connect different elements. Similar to the meaning of edge in PMR graph, the meaning of EPC *arc* is determined by the type of source node and target node. Hence, the *arc* in EPC is mapped to the edge in PMR graph, which is shown in R1.
2) Trigged by one or more events producing one or more new events, *Function* in EPC represents a specific activity, which corresponds to *process node* in PMR Graph. So, *Function* in EPC is mapped to *process model* in PMR graph, as shown in rule R2.
3) *Connector* in EPC model can be mapped into different kinds of nodes in PMR graph according to its in-degree and out-degree. For the *split dependency node* and *join dependency node*, the logic type of *Connector* such as XOR, AND, and OR is transferred to the logical type Typ in PMR graph. Constrained by the definition of structural requirements of PMR graph, a new *split option node* or *join option node* is added into each branch after finishing *Connector* map, as described by R3 and R4. The detailed mapping procedure for *Connector* is shown in Algorithm 2.

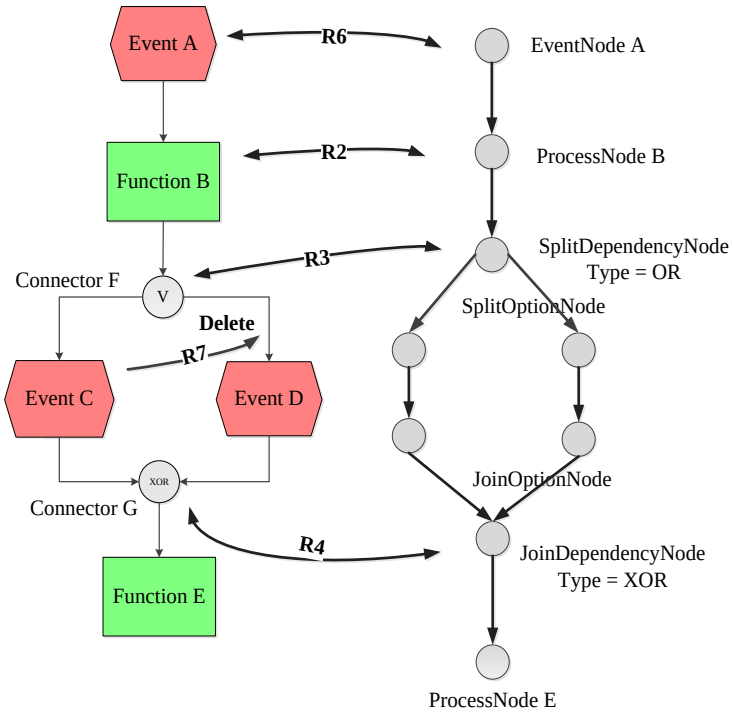


Fig. 5. Mapping rules for core elements.

4) If the previous node and subsequent node of an *Event* are *Functions*, the *Event* is converted to *sequence node* in PMR graph, which is described by rule R5. If an *Event* has no previous node or has no subsequent node, it is mapped into event node in PMR graph, working as start node or end node of PMR Graph, which is described in rule R6. If the previous node of *Event* is *Connector* and there are one or more subsequent nodes, the *Event* is not mapped into any node in PMR graph and is removed, which is shown in rule R7. It is noted that all the text labels contained in the *Event* will be *kept* and *transferred* into corresponding nodes in PMR Graph, and how to transfer them is left to the Section 4.2.2.

A process fragment is shown in Fig. 5, which exemplifies how to map the core elements of EPC model to those in PMR graph. First, according to R3, the OR connector in EPC model is mapped to *split dependency node* of type OR. At the same time, a new *split option node* is added for each of two branches. Similarly, the XOR connector in EPC model is mapped into *join dependency node* as well as a new *join option node* according to R4. Then, we obtain a *split option node* connected directly to a *join option node* at each branch. Next, Event A is mapped to the start node of the PMR graph through R6. Lastly, in terms of R7, Event C and D are deleted due to their previous nodes are connectors and there is another connector to the subsequent node.

4.2.2 Mapping Attributes of Elements

After using the rules in Table 1 to map the core elements, a skeleton of PMR graph is obtained. However, as a generic phase, part of the attributes will need to be redistributed and adjusted after the core elements have been mapped from a specified process model to PMR graph.

The meta-rule for mapping attributes from a language-specific process model to PMR graph is depicted below.

$$Element(X) \wedge Predecessor(Y) \wedge Attr(Y, y) \wedge Successor(Z) \wedge Attr(Z, z) \wedge Parent(W) \wedge Attr(W, w) \wedge ElementInPMR(V) \wedge Attr(V, v) \wedge M(X, V) \Rightarrow Transfer(y, v) \vee Transfer(z, v) \vee Transfer(w, v)$$

The rule describes that a process model element X in a language-specific process model corresponds to an element V in PMR graph. v is one of attributes of V. If Y is the predecessor of X, Z is the successor of X, W is the parent of X, and y, z, w are attributes of Y, Z, W, then the attribute y, or the attribute z, or the attribute w is mapped to the attribute v.

With regard to EPC model, the attribute mapping rules are shown in Table 2, which are applied to fill the PMR graph with text labels existed in *event* of original EPC model. Text labels contained in *event* will be mapped into a set of pre-conditions and effects associated with a process node after the core elements have been aligned between the EPC model and PMR graph. In Table 2, the column "Elements connected with Event" describes the different situations categorized by the successive or previous nodes of the *event* in EPC model, while the column "PMR graph node or properties" represents the corresponding elements in PMR graph after mapping *event* and all the other core elements adjacent to *event* to PMR graph and completing text label propagation.

R8 describes a mapping scenario where the subsequent node of an event is a function. In this scenario, the event triggers the function while the precondition of a process node describes the constraints that must be true before a process node is invoked. The text label of the Event is mapped to the precondition of the aligned process node due to the semantic equivalence between them. A similar mapping scenario is described in R9, where the previous node of an event is a function.

R10 and R11 describe the scenarios where the previous or subsequent node of an event is a connector. To facilitate the mapping of an event text label, a property named *transitional expression* is designated and associated with a dependency node in PMR graph for saving the event text label temporarily. With the help of this property, the event text label could be mapped from the event to the process nodes that the nearest function corresponds to.

A process fragment shown in Fig. 6 is used to exemplify how to map the *event* text labels. According to R10, the text label of Event B is mapped to the upper split dependency node and the lower join dependency node, in which the transitional expression property is harnessed to save the text label of Event B. Next, the text labels for Event C and D are propagated to the upper split dependency node and the lower join dependency node and saved temporarily under the guidance of R11, as shown in Fig. 6.

TABLE 2. MAPPING RULES FOR ELEMENT ATTRIBUTES

Rule No.	*Elements that attribute propagates to*	*Propagation of label*
R8	Function (Predecessor: Event)	M(Function). precondition ← Event. name
R9	Function (Successor: Event)	M(Function). effect ← Event. name
R10	Connector (Predecessor/Successor: Event and M(connector).TransitExp is empty)	M(connector). TransitExp ← Event. name
R11	Connector (Predecessor/Successor: Event and M(connector).TransitExp is not empty)	M(connector). TransitExp. append(Connector. Type + Event. name)

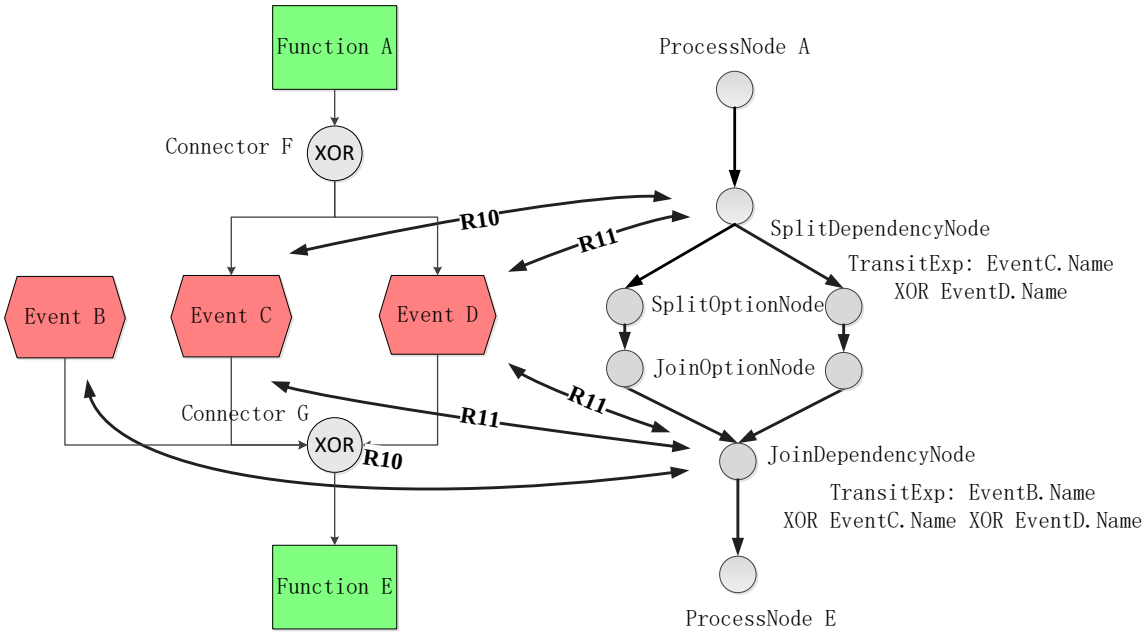


Fig. 6. Mapping attributes of text label of Event.

4.2.3 Transformation Algorithm

Before we present the transformation algorithms, we need to define the mapping function M that transforms a node in PST to a node in PMR graph.

Definition 2 (Mapping Function *M*). Let F be a set of nodes of a PMR graph and $Basic$ a set of nodes a process structure tree PST. The mapping $M: Basic \rightarrow F$ defines a transformation of a node of PST to a node in PMR graph.

Algorithm 1 for this phase takes a PST as input and returns a PMR graph. The *general idea* of our strategy is to recursively traverses the nested structure of a PST in a top-down manner. The input is the root of this PST. The output is the corresponding PMR graph to the PST. The main procedure is described as follows. First, all of the children of the root are populated to queue q. Then, all the PST nodes in q are traversed. If a PST node is leaf node of PST, which means it cannot be further decomposed, it is transformed to a process element in PMR graph according to a specified transformation rule. However, if a PST node is a non-leaf one, which means it can be further decomposed, the function $PSTTransform$ will be recursively called with this non-leaf node as the root. The recursive call will terminate after all the non-leaf nodes in the PST have been traversed.

The procedure $transformPSTNode$ used in the $PSTTransform$ procedure generates the PMR graph element that corresponds to the respective process element in language-specific process model. This procedure conducts the concrete transformation between language-specific process models to PMR graphs. In the following, we give two concrete examples, i.e., $transformConnector$ and $transformEvent$, which help transform Connector and Event in a EPC model to the corresponding elements in PMR graph. The last step is to transfer the label text on the path of Dependency Nodes to the neighboring process nodes and then update the PMR graph.

Algorithm 2 shows how to map a *connector* in EPC model into a *dependency* node in PMR graph. Line 1-2 obtains all the previous and successive nodes of a *connector*. Line 3-12 shows how a *connector* is mapped into a *split dependency* node in PMR graph. Line 4 creates a new *split dependency* node, and the logical types or this split dependency node is assigned in Line 5. Line 7-11 connects *split dependency* nodes with its previous nodes and successive nodes, which are created by means of mapping rules described in Table 1. Similarly, the join dependency node is created from Line 13 to Line 21.

Algorithm 3 describes how to map an *event* in EPC model into the corresponding node in PMR graph. First, the position where an *event* lies in is judged and categorized into three different situations. Then, mapping rules are invoked to create new nodes and edges in PMR graph in terms of mapping rules described above. The previous or successive nodes for the *event* are obtained in Line 1-2. The *event*

Algorithm 1. Pseudo code for PSTtransform(PSTNode $root$, PST pst, $PMRGraph\ pg$)

Input: $PSTNode\ root, PST\ pst$
Output: $PMRGraph\ pg$
01. $q \leftarrow$ empty queue
02. q.enqueue($root$.getAllChildren())
03. **while** (not q.isEmpty())
04. $node \leftarrow q$.dequeue()
05. **if** $node \in leafNode$ in pst **then**
06. $pg \leftarrow$ transformPSTNode($node, pst$)
07. **end if**
08. **else if** $node \in$ non leafNode **then**
09. $pg \leftarrow$ PSTtransform($node, pst, pg$)
10. **end if**
11. **end while**
12. **For each** Dependency Node Chain p in pg
13. $pg \leftarrow GetTransitExp(pg, p)$
14. **end for**

without previous nodes or successive nodes is mapped to the newly created event node in PMR graph. The text label in the *event* is mapped to the previous or successive nodes in terms of the rules described in Table 2. If both the previous and successive node of an *event* are functions, the *event* is converted to *sequence* node in PMR graph, and the text label of the *event* is mapped to the *process* nodes that its adjacent *function*s correspond to in terms of R8 and R9 (Line 9-14). Line 15-17 illustrates a situation that the previous node of an *event* is a *connector* and the successive node is a *function*. In this case, the event is totally removed, while the text label is mapped to the process node of that its successive *function* corresponds to.

4.2.4 Propagation for Text Label of Event

So far we have mapped the attributes of elements into the most directly mapped elements in PMR graph. This, however, does not necessarily lead to a correct solution in terms of the definition of PMR graph. In fact, after mapping *node*s and *event*s from EPC model to PMR graph, the text label of some *event*s are still kept in the *Split/Join Dependency Nodes* as a transitional expression, which needs to be transferred to the nearby *process* nodes eventually.

Fig. 7 illustrates another example with regard to this point. Here H, I, J are dependency nodes partially mapped to connector H, I, and J in EPC model. The transitional expression of J is the union set of name of Event E, F, and G. The transitional expression of I is any one of names of Event C, D, and transitional expression of J, while the transitional expression of H is the combination of name of Event node B and the transitional expression of dependency node I. These transitional expressions temporarily stored in the dependency node represent a state after an activity is finished and must be propagated to the closest process node as the *effect* attribute.

If we analyze this phenomenon further, we observe that the underlying cause is the existence of a chain of dependency nodes, which obstruct the attribute propagation from event to process node. Specially, the attribute of Event F in the left graph is firstly mapped to the transitional expression of Dependency node J. Yet, there are other two Dependency nodes H and I, which need to be stepped over for the attribute propagation from Dependency J to Process Node A.

To overcome this situation, we define four other rules to describe attribute propagations between a pair of neighbor nodes A and B in PMR graph. The rules are shown in Table 3.

R12 and R13 describe the scenario that another *Dependency Node* named B is the previous/successive element of the *Dependency Node* named A. If there is no *transitional expression* in the *Dependency Node*, then *transitional expression* of *Dependency Node* A is propagated into *Dependency Node* B, which is described in rule R12. On the other hand, if *transitional expression* of *Dependency Node* B is not empty, transitional expression of *Dependency Node* A is propagated to the *Dependency Node* B, and then appended to the *transitional expression* of *Dependency Node* B through the logical type of *Dependency Node* B, constituting a new composite *transitional expression*, which is illustrated in rule R13.

Rules R14 and R15 illustrate the scenario that *Process Node* is the previous/successive element of *Dependency Node*. In these scenarios, the transitional expression is propagated to the *Process Node*, and saved as effect/precondition property in the process node.

Algorithm 2. ***transformPSTNode(Connector c, PST pst, PMRGraph pg)***

Input: Connector c, PST pst, PMR graph pg
Output: Updated PMR Graph pg

01. NodeSet $N_p \leftarrow pst.c.predecessor$
02. NodeSet $N_s \leftarrow pst.c.successor$
03. **If** the number of $N_p = 1$ and the number of $N_s > 1$ **then**
04. Create Split Dependency Node SDN
05. The type of $SDN \leftarrow$ the type of c
06. Connect $M(N_p)$ to SDN
07. **For each** Node n_s in N_s
08. Create Split Dependency Option Node $SDON$
09. Connect SDN to $SDON$
10. Connect $SDON$ to $M(N_s)$
11. **End for**
12. **End if**
13. **If** the number of $N_p > 1$ and the number of $N_s > 1$ **then**
14. Create Join Dependency Node JDN
15. The type of $JDN \leftarrow$ the type of c
16. **For each** Node n_p in N_p
17. Create Join Dependency Option Node $JDON$
18. Connect $M(n_p)$ to JDN
19. Connect JDN to $JDON$
20. **End for**
21. **End if**
22. Return pg

Algorithm 3 ***transformPSTNode(Event e, PST pst, PMRGraph pg)***

Input: Event e, PST pst, PMR graph pg
Output: Updated PMR Graph pg

01. NodeSet $N_p \leftarrow pst.e.predecessor$
02. NodeSet $N_s \leftarrow pst.e.successor$
03. **If** the number of $N_p = 0$ or the number of $N_s = 0$ **then**
04. Create Event Node EN
05. Add EN into pg
06. Connect $M(N_p)$ to EN in pg
07. Connect EN to $M(N_s)$ in pg
08. Invoke R8, R9, R10, R11 to propagate text label in e
09. **Else if** N_p is Function and N_s is Function **then**
10. Create Sequence Node SQN
11. Add SQN into pg
12. Connect $M(N_p)$ to SQN in pg
13. Connect SQN to $M(N_s)$ in pg
14. Invoke R8, R9 to propagate text label in e
15. **Else if** N_p is Connector and the number of $N_s = 1$ **then**
16. Connect $M(N_p)$ to $M(N_s)$ in pg
17. Invoke R10, R11 to propagate text label in e
18. **End if**
19. Return pg

Let N_0 be the start point of a chain of *Dependency Nodes* expressed as $p = \{N_0, N_1, N_2, \dots, N_m\}$. TE_i is the transitional expression of the node N_i of the dependency node chain. The transitional expression of N_0 could be obtained by the function $GetTransitExp()$ in Algorithm 4. This algorithm starts by calling R12 or R13 shown in Table 3 to obtain the transitional expression from the last Dependency Node N_m to the start point N_0 of p, obtaining the transitional expression of the start point N_0, which will be transferred to the neighboring process node as $Postcondition$. A similar process can be employed to obtain $Precondtion$ for a process node from a chain of Dependency Nodes.

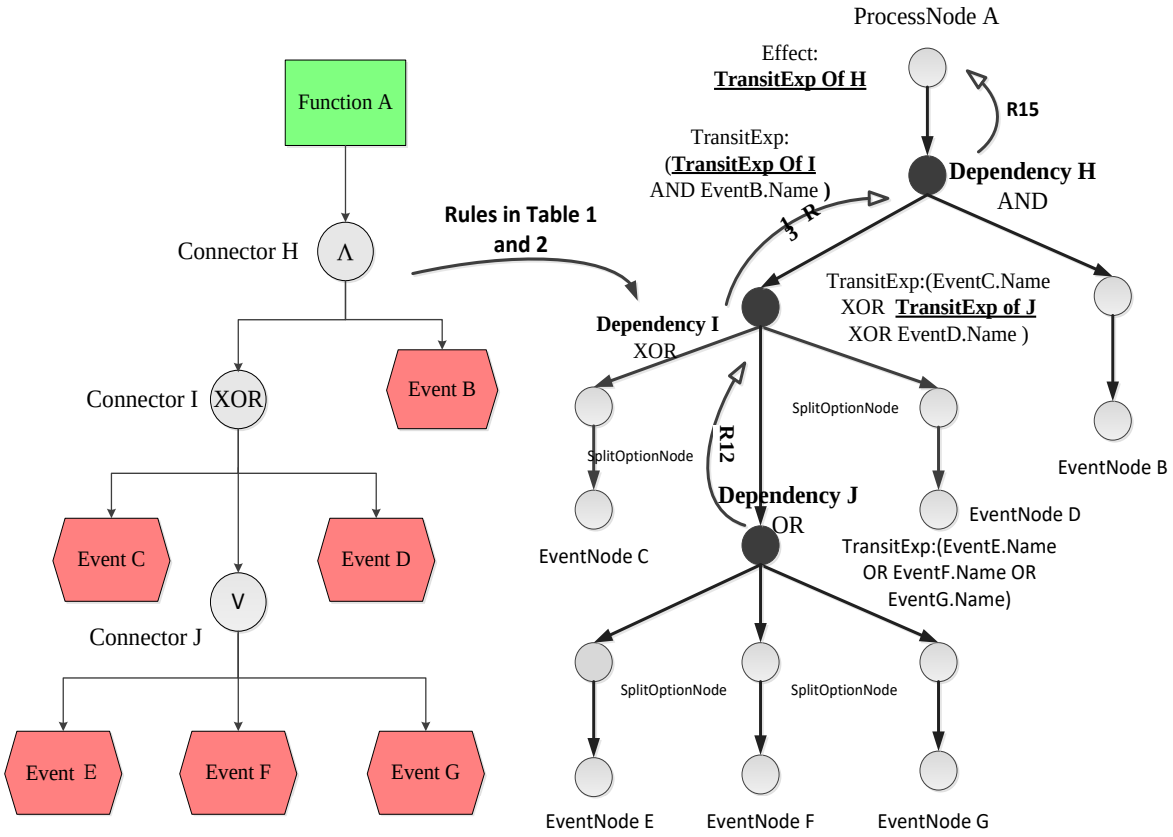


Fig. 7. Propagation for text label of Event in the chain of dependency nodes.

Table 3. PROPAGATION RULES FOR LABELS INSIDE PMR GRAPH

Rule No.	*Elements that attribute propagates to*	*Propagation of label*
R12	Dependency Node (Predecessor/Successor: another Dependency Node and Dependency Node.TransitExp is empty)	DependencyNode. TransitExp ← Predecessor/Sucessor. TransitExp
R13	Dependency Node (Predecessor/Successor: another Dependency Node and Dependency Node.TransitExp is not empty)	DependencyNode. TransitExp. append(DependencyNode. Type +Predecessor/Sucessor. TransitExp)
R14	Process Node (Predecessor: Dependency Node)	ProcessNode. Precondition ← DependencyNode. TransitExp
R15	Process Node (Successor: Dependency Node)	ProcessNode. Postcondition ← DependencyNode. TransitExp

Algorithm 4. ***GetTransitExp(PMR Graph pg, Dependency Node Chain p)***

Input: PMR graph pg, Dependency Node Chain p
Output:Updated PMR graph pg
1. Let N_0 be the start point of $p = \{N_0, N_1, N_2, \dots, N_m\}$
2. **foreach** Dependency Node N_i ordered from N_m to N_0
3. call R12 or R13 to obtain TE_i
4. **end for**
5. call R14 or R15 to transfer lable text from N_0 to the neighbouring process node
6. Return updated pg

The right graph of Fig. 7 exemplifies how to propagate the attributes in the Dependency node chain. According to R12, the transitional expression of Dependency node J is transferred and integrated into the transitional expression of Dependency node I combining with the attributes from Event E, F, and G.

After that, similarly, the transitional expression of the Dependency node I is transferred and integrated into the transitional expression of the Dependency node H in terms of rule R13. Eventually, according to rule R15, the transitional expression of the Dependency node H is transferred and becomes the *effect* of Process Node A. By this means, the text label attribute of the Dependency node chain propagate progressively until they become the precondition or effect of functions of PMR graph.

4.2.5 Reduction Rule

After transforming a language-specific process model into a PMR graph, we can simplify the resulting graph by applying the reduction rules. This rule is designed to eliminate "unnecessary" split option nodes or split join nodes. The rules are applied until a PMR graph cannot be further reduced.

Function $MergeConsecutiveOptionNodes$ is a *cleaning operation* that merges two consecutive option nodes into a single split (join) option node between a pair of split/join dependency nodes. Here, one of the option node is redundant and can be removed if a *split/join option node* is connected directly to another *join/split option node*. As shown in Fig. 8, an arbitrary one of these two option nodes should be removed to avoid redundancy. Here, we remove the *split option nodes* C and H, while keeping the *join option nodes* D and F.

4.3 Complexity Analysis

The algorithm for transforming a PST to the PMR graph is a recursive algorithm. The number of recursive calls depend on the number of sub-processes in the language-specific process model. Suppose there exist m sub-processes in the process model, the complexity of the transforming algorithm is $mT(n)$, where $T(n)$ represents the complexity of transforming a process model in which all the process elements are at the same level.

The complexity of the algorithm for transforming connectors (Algorithm 2) is linear on the maximum number of predecessors/successors of each connector. The number of connectors is bounded by the number of the nodes ϵ of the language-specific process model, while the maximum number of predecessors/successors of a connector is bounded by the maximum degree δ among of nodes in the language-specific process model. Thus the complexity of transforming all of connectors is $O(\epsilon\delta)$.

The complexity of the algorithm for transforming an event (Algorithm 3) is $O(1)$. Thus the complexity of transforming all of events in a process model is bounded by the number of the nodes ε, which is $O(\varepsilon)$. The complexity of algorithm for calculating label transfer on a dependency node path is linear on the number of dependency nodes on the chain of dependency nodes. The number of dependency nodes is bounded by the number of the nodes ε of the language-specific process model. Thus the complexity of Algorithm 4 is $O(\varepsilon)$. Hence, the complexity of the whole transformation algorithm is $m(\delta + 2)O(\epsilon)$.

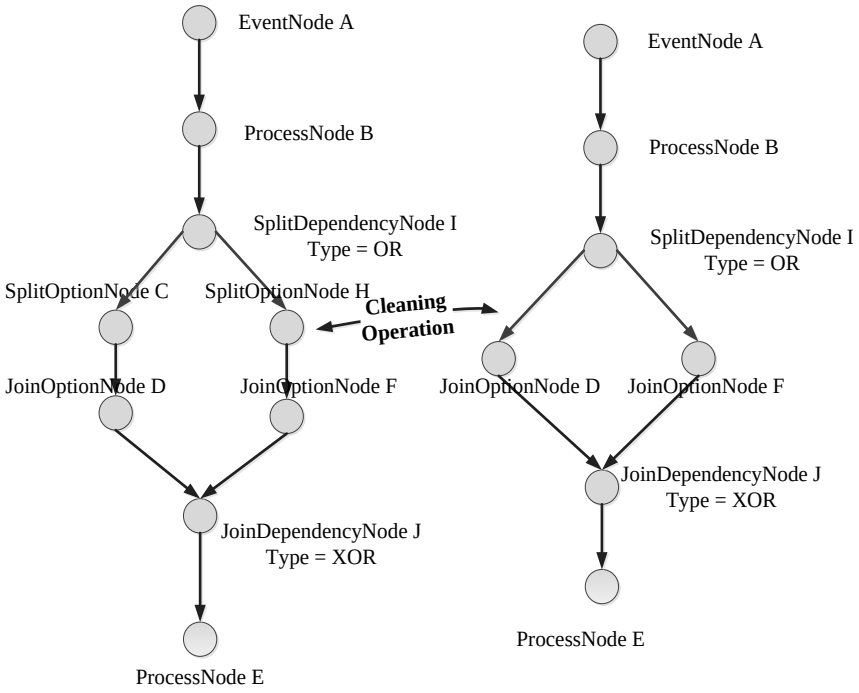


Fig. 8. Cleaning Operation.

4.4 Property of the Algorithm

In Section 1, we stated that the algorithm should satisfy the requirement of behavior-preservation. Below, we sketch the proof of this proposition showing that the algorithms fulfill the requirement of behavior-preservation.

Proposition 1. *Let PG be the PMR graph that is converted from an EPC model via using Algorithms 1, 2, 3, and 4. Let EG represent the graph structure of the EPC model. Any execution trace of EG has the identical mapping in PG, and all execution traces of EG can be mapped to PG. That is to say, PG and EG have the same execution traces.*

Proof. Let $e_1, e_2, \ldots, e_n$ be an execution trace of EG, which is represented as a sequence of edges. Taking an edge $e_i = (k, n)$ as an example, here are a couple of situations that we should consider if e_i is a subset of edge set of PG.

(1) There is no adding or removal of nodes during the process of edge mapping - For instance, if k is an event and n is a function, according to the lines 3~7 of the Algorithm 3, there should be a node k' in PG, that is transformed from the event k, and k' is connected to its process node n', which is transformed from the function n. According to R2, the process node in PG is corresponding to the function in EG. Accordingly, the edge (k', n') corresponds to the edge (k, n) in EG. Similarly, for the edge (k, n), if k is a function and n is an event, we can arrive at the same conclusion.

(2) There exist adding or removal of nodes during the process of edge mapping from EG to PG - For instance, if k is an OR-split connector and n is an event. According to lines 3~7of the Algorithm 2, this connector is mapped to a Split Dependency Node k' and a couple of Option Nodes n_1', n_2',…, n_j'. Further, in terms of lines 15~17 of the Algorithm 2, the event is removed while the text label of this event is transferred and stored into the Option Nodes n_1', n_2',…, n_j'. So the edge (k, n) is corresponding to the edges (k', n_1'), (k', n_2'),…, (k', n_j'). For other similar situations, the same conclusions can be obtained.

(3) If k is a split option node and n is a join option node - In this case, cleaning operations remove the redundant edge between split option node and join option node. However, there is no corresponding edge of this kind of redundant edge in EG. So the execution trace is preserved since the removed edge (k, n) does not matter.

In summary, any execution trace in EG is also an execution trace of PG. Accordingly, our proposed mapping algorithm can preserve the behavior of EPC model.

5 EVALUATION

The mapping algorithms from different process languages to PMR metamodel has been implemented as a tool, namely *Process Graph Generator*, that is freely available as part of the PMRMP toolset [23].[1] So far, the tool supports conversion of EPC represented in the EPML format, BPEL [13], and OWL-S [2] process model represented in OWL format. BPEP allows users to edit process models in a variety of languages (such as EPCs, BPEL, and OWL-S Process) through the Web portal. Different kinds of process models can be transformed into PMR graph as the output, which can be further stored in a relational database.

Using the implementation of the algorithm, we conducted experiments to evaluate the correctness and the performance of our mapping algorithms from the perspective of EPC conversion. Furthermore, we conducted a case study to evaluate the usefulness based on a real-world repository. The tests were conducted on a laptop with a quad-core Intel processor, 2.7 GHz, 4GB memory, running Microsoft Windows 10 64-bit.

5.1Correctness of the algorithm

We evaluate the percentage of EPC models that can be converted to PMR graph by means of our mapping rules and algorithms in order to verify the correctness of our algorithm. We conducted the experiments on the real-world SAP R/3 reference model [24], which contains 604 models with sizes ranging from 5 to 119 nodes. Our experiment result shows that 573 models of them can be converted successfully. The success rate is: 573/604 = 94.9%. The rest of EPC models failed to be converted due to the violation of the EPC well-formedness constraint defined by Rosemann and van der Aalst [4]. For example, there are some models with connectors possessing multiple inputs and multiple outputs, which break the constraint of a legal EPC model.

[1] Available at: https://github.com/Zaiwen/PMRMP

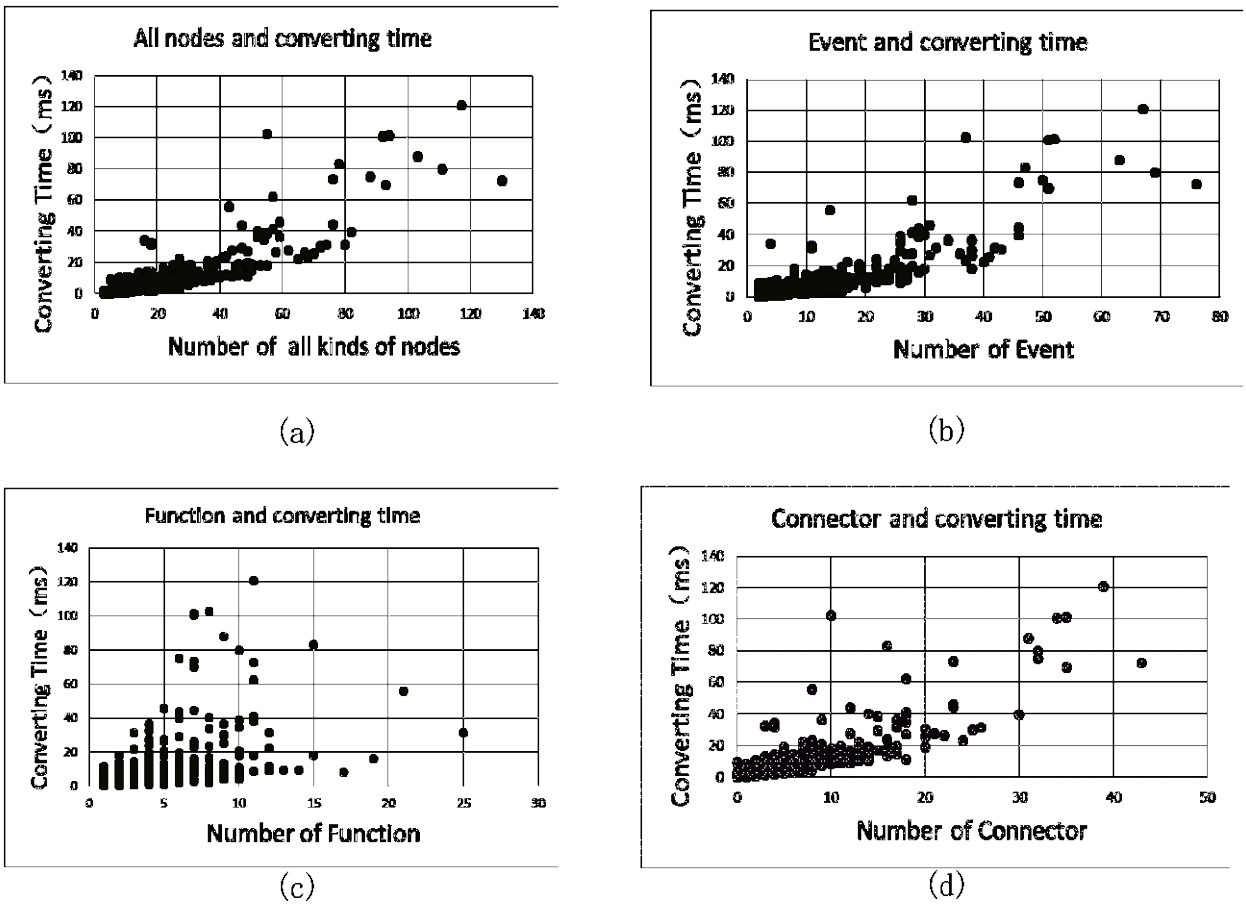


Fig. 9. The response time related to (1) Number of all kinds of nodes (2) Number of events (3) Number of functions (4) Number of connectors.

5.2 Time performance

This experiment aims to evaluate the time performance of the algorithm and analyze the relationship between the response time and various types of nodes in EPC models. We compute the number of all types of nodes including *event*, *function*, and *connector*, recording the response time from converting EPC model to PMR graph for each of legitimate EPC model at the same time. We obtain the results shown in Fig. 9.

As shown in Fig. 9(a), we can conclude that the response time is positively correlated with the number of nodes in EPC. It is possible for some models with less number of nodes to consume more processing time, since the distribution of *node type* is uneven and the processing costs for *connector* and *event* are more than *function*. From Fig. 9(b), we can see that the number of events in EPC model takes a significant positively effect on the response time. Simlilarly, the reponse time are highly positively related with the number of connectors in EPC model, which is shown in Fig. 9(d). Different from *event* and *connector*, when the number of *function* increases, the response time does not show a distinct incremental trend, indicating that the number of *function* has no obvious effect on the response time, which is shown in Fig. 9(c).

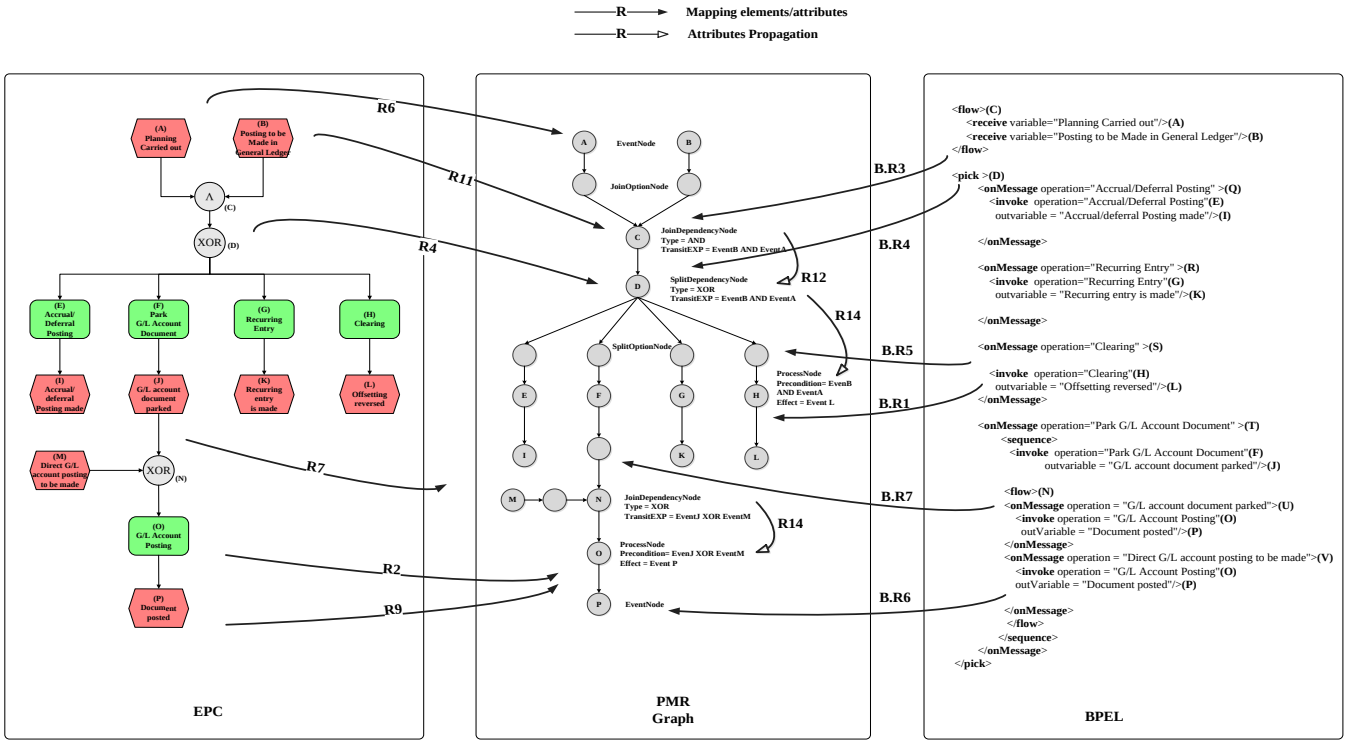


Fig. 10. A sample scenario for integrating different models into PMR

5.3 Case study

In order to further illustrate the operational steps of integrating different process models into PMR and prove the generalization of the proposed registering framework, we take a real financial business process model from SAP R/3 reference model for another case study. The process model is initially in the format of EPML. However, a BPEL model with similar semantics of the EPC model is built by a domain expert after understanding the meaning of EPC model thoroughly. Then, the same registration framework is followed to map the dual process models into PMR graph.

As we point out in Section 4.2, different mapping rules are needed for transforming different language-specific process model into PMR graph even though the general meta-rules exist and can be

TABLE 4. MAPPING RULES FOR BPEL ELEMENTAND ATTRIBUTES

Rule No.	*BPEL Elements*	*PMR Graph Elements*
B.R1	bpel:invoke	Process
B.R2	bpel:invoke/@operation	Process.name = operation
B.R3	bpel:flow	Split_Dependency and Join_Dependency
B.R4	bpel:*pick*	Split_Dependency and Join_Dependency
B.R5	bpel:pick/bpel:onMessage	Split_Dependency_OptionJoin_Dependency_Option
B.R6	bpel:invoke/@outputVariable	The association between ProcessNode and EventNode EventNode.name = outputVariable.value
B.R7	bpel:flow/bpel:onMessage	Split_Dependency_OptionJoin_Dependency_Option

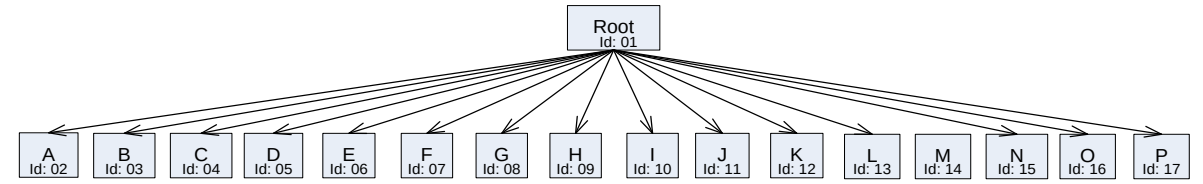
used to advise the generation of language-specific mapping rules. We reuse part of the mapping rules from BPEL to PMR described in our previous work [25]. These mapping rules are shown in Table 4.

Step 1: Model Decomposition

The EPC model and BPEL model are firstly decomposed into two PSTs respectively. For EPC model, only one leaf layer exists in PST because there is not any sub-process in the model. Each leaf node represents an EPC model element in this PST, while at the same time corresponds to a language-specific manageable atom. Every manageable atom records the predecessor(s) and successor(s) of each EPC

modeling element. For instance, the node N (id: 15) is a XOR connector. It has two predecessors which are Event J (id: 11) and M (id: 14), and one successor which is Function O (id: 16).

The PST that BPEL model is decomposed into is shown in Figure 11. Comparing to PST of EPC model, it has multiple non-leaf layers because BEPL is categorized into a block-structured process modeling language and each non-leaf node represents a structural activity in BEPL model. For instance, the node S (id: 08) represents a structural activity$< onMessage >$ and it contains another block named$H$ (id: 12), which stands for the structural activity$< invoke >$.



Process	
ProcessId	1
Language	EPC
PMEId	01,02,03,04,05, 06,07,08,09,10, 11,12,13,14,15, 16,17

Process Model Element	
PMEId	01
PMEType	Root
PMEDesc	Root
Parent	NULL
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	02
PMEType	Event
PMEDesc	A
Parent	01
Predecessor	NULL
Successor	04

Process Model Element	
PMEId	03
PMEType	Event
PMEDesc	B
Parent	01
Predecessor	NULL
Successor	04

Process Model Element	
PMEId	04
PMEType	AND
PMEDesc	C
Parent	01
Predecessor	02,03
Successor	05

Process Model Element	
PMEId	05
PMEType	XOR
PMEDesc	D
Parent	01
Predecessor	04
Successor	06,07,08,09

Process Model Element	
PMEId	06
PMEType	Function
PMEDesc	E
Parent	01
Predecessor	05
Successor	10

Process Model Element	
PMEId	07
PMEType	Function
PMEDesc	F
Parent	01
Predecessor	05
Successor	11

Process Model Element	
PMEId	08
PMEType	Function
PMEDesc	G
Parent	01
Predecessor	05
Successor	12

Process Model Element	
PMEId	09
PMEType	Function
PMEDesc	H
Parent	01
Predecessor	05
Successor	13

Process Model Element	
PMEId	10
PMEType	Event
PMEDesc	I
Parent	01
Predecessor	06
Successor	NULL

Process Model Element	
PMEId	11
PMEType	Event
PMEDesc	J
Parent	01
Predecessor	F
Successor	15

Process Model Element	
PMEId	12
PMEType	Event
PMEDesc	K
Parent	01
Predecessor	08
Successor	NUL

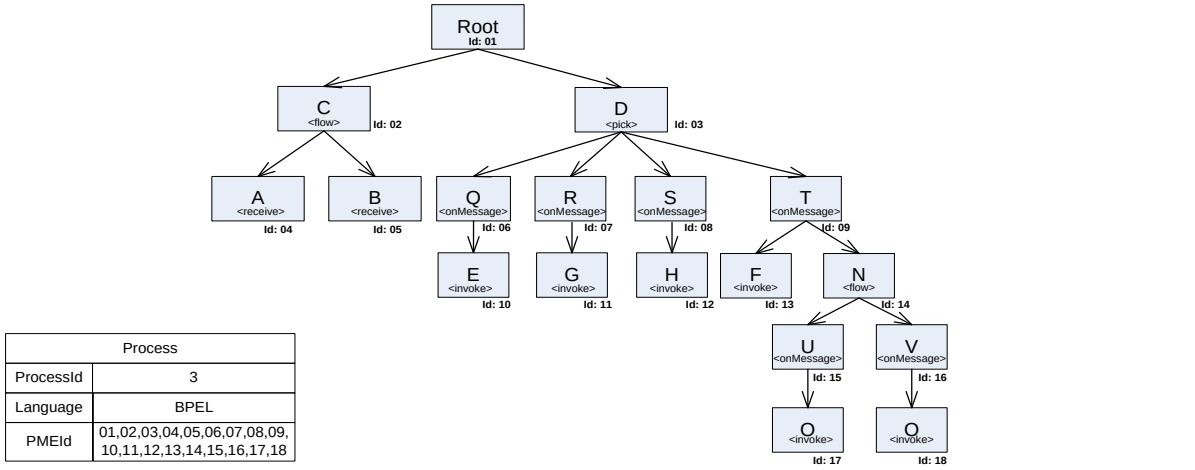
Process Model Element	
PMEId	13
PMEType	Event
PMEDesc	L
Parent	01
Predecessor	09
Successor	NULL

Process Model Element	
PMEId	14
PMEType	Event
PMEDesc	M
Parent	01
Predecessor	NULL
Successor	15

Process Model Element	
PMEId	15
PMEType	XOR
PMEDesc	N
Parent	01
Predecessor	11,14
Successor	16

Process Model Element	
PMEId	16
PMEType	Function
PMEDesc	O
Parent	01
Predecessor	15
Successor	17

Process Model Element	
PMEId	17
PMEType	Event
PMEDesc	P
Parent	01
Predecessor	16
Successor	NULL



Process	
ProcessId	3
Language	BPEL
PMEId	01,02,03,04,05,06,07,08,09, 10,11,12,13,14,15,16,17,18

Process Model Element	
PMEId	01
PMEType	Root
PMEDesc	Root
Parent	NULL
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	02
PMEType	<flow>
PMEDesc	C
Parent	01
Predecessor	NULL
Successor	03
Nested	FALSE

Process Model Element	
PMEId	03
PMEType	<pick>
PMEDesc	D
Parent	01
Predecessor	02
Successor	NULL
Nested	FALSE

Process Model Element	
PMEId	04
PMEType	<receive>
PMEDesc	A
Parent	04
Predecessor	NULL
Successor	05

Process Model Element	
PMEId	05
PMEType	<receive>
PMEDesc	B
Parent	05
Predecessor	04
Successor	NULL

Process Model Element	
PMEId	06
PMEType	<onMessage>
PMEDesc	Q
Parent	03
Predecessor	NULL
Successor	07
Nested	FALSE

Process Model Element	
PMEId	07
PMEType	<onMessage>
PMEDesc	R
Parent	03
Predecessor	06
Successor	08
Nested	FALSE

Process Model Element	
PMEId	08
PMEType	<onMessage>
PMEDesc	S
Parent	03
Predecessor	07
Successor	09
Nested	FALSE

Process Model Element	
PMEId	09
PMEType	<onMessage>
PMEDesc	T
Parent	03
Predecessor	08
Successor	NULL
Nested	FALSE

Process Model Element	
PMEId	10
PMEType	<invoke>
PMEDesc	E
Parent	Q
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	11
PMEType	<invoke>
PMEDesc	G
Parent	R
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	12
PMEType	<invoke>
PMEDesc	H
Parent	S
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	13
PMEType	<invoke>
PMEDesc	F
Parent	T
Predecessor	NULL
Successor	14

Process Model Element	
PMEId	14
PMEType	<flow>
PMEDesc	N
Parent	T
Predecessor	13
Successor	NULL
Nested	FALSE

Process Model Element	
PMEId	15
PMEType	<onMessage>
PMEDesc	U
Parent	N
Predecessor	NULL
Successor	16
Nested	FALSE

Process Model Element	
PMEId	16
PMEType	<onMessage>
PMEDesc	V
Parent	N
Predecessor	15
Successor	NULL
Nested	FALSE

Process Model Element	
PMEId	17
PMEType	<invoke>
PMEDesc	O
Parent	U
Predecessor	NULL
Successor	NULL

Process Model Element	
PMEId	18
PMEType	<invoke>
PMEDesc	O
Parent	V
Predecessor	NULL
Successor	NULL

Fig. 11. PST and the common language-specific manageable atoms of the EPC and BPEL model in Fig. 10.

Step 2: Mapping elements

As Fig. 10 shows, the elements in the EPC model are converted into structure of PMR graph based on the rules in Table 1. Three Connectors, named C, D, and N, are mapped to *Split Dependency Nodes* and *Join Dependency Nodes* in PMR graph accordingly by using the rule R3 and R4. Seven Events, named A, B, I, K, L, M, P, are mapped to *Event Nodes* in PMR graph under the guideline of R6, while Event I is removed according to R7. Five Functions, named E, F, G, H, and O, are mapped into *Process Node*s according to R2.

The BPEL process model with the same semantics is shown in the right side of Fig. 10. The corresponding elements between EPC and BPEL are aligned with the same alphabet. Five *invoke* activity E, F, G, H, O are mapped to *Process Node*s in PMR graph referring to the rule B.R1. Two *flow* elements C and N are mapped to *Split Dependency Node* and *Join Dependency Node* according to B.R3. The *pick* element D is mapped into *Split Dependency Node* by using B.R4. The *on-message* elements Q, R, S, and T are mapped into *Split Option Node* according to B.R5 while the other *on-message* elements U, V are mapped into *Join Option Node* according to B.R7.

Step 3: Mapping attributes

As described in Section 4.2.2, the attributes of elements such as *event* name and *function* name must be assigned to the node of PMR graph when EPC model is mapped into PMR graph. For instances, the name of Event A and B are mapped into the transitional expression of *Join Dependency Node* C in PMR graph according to R10 and R11. The name of Event P will be assigned to the effect of *Process Node* according to R9. With regard to the mapping of BPEL process, the operation field of *invoke* activity is mapped to the name of *Process Node* in PMR Graph by using B.R2. The *out variable* of *invoke*, such as element P, is mapped to the name of *Event Node*.

The intermediate PMR graph is created after the elements and attributes are mapped from a specified model into PMR graph. This step aims to adjust and redistribute some attributes inside the PMR graph. As shown in Fig. 10, the *transit Exp* of *Split Dependency Node* D is empty before attribute propagation. The *transit Exp* of previous *Join Dependency Node* C, namely *Event* A and *Event* B, is propagated to D according to the rule R12. The precondition of *Process Node* H is empty and then assigned to be *transit Exp* of *Join Dependency* D according to the rule R14. The similar situation happens on the propagation of *transit Exp* of *Join Dependency Node* N to precondition of *Process Node* O.

Following the same procedure defined in the registering framework, two process models in Fig. 10, though they are described by the different process languages, are mapping into the same PMR graph. This case study proves that our framework and procedures are generic and could be applied for different kinds of process languages if corresponding mapping rules are provided.

6 Conclusion and Future Work

In this article, we proposed a generic registration framework that maps a process model in a specific process language to PMR registration item. Considering Event-driven Process Chain (EPC) is a kind of popular process model that is widely used in industry, we focus on the mapping rules and related algorithm from EPC to PMR graph and develop an automatic process model registration tool for EPC. The interoperation capability of the PMR metamodel was evaluated in our article. In addition, based on SAP EPCs, we conducted experiments to demonstrate the feasibility and performance of our approach. The experiments showed that, first of all, our proposed registering framework is generic for mapping different kinds of process languages to PMR. Moreover, our approaches could realize the mapping from EPC models to PMR graph within 100ms. The response time for mapping EPC model to PMR graph has a positive correlation with the number of all nodes, connectors and events. The results indicate that our approach provides a solid foundation and infrastructure for the modeling and execution of adaptable processes for enterprise collaboration.

In the future, we plan to complement adaption operations based on PMR graph further, summarize adaption operations based on EPC models, and set up the change propagation operations between EPC models and PMR graph. For real-life applications, we are planning to apply our study into different real-life applications of tourism [16] and workforce management [17].

Acknowledgement

This work is supported by the National 973 Basic Research Program of China under Grant

No.2014CB340404, the National Natural Science Foundation of China under Grant 61100017,61562073, and 61672387. We would like to appreciate the anonymous reviewers for the valuable comments.

REFERENCE

1. Davies, I., Green, et al (2006). How do practitioners use conceptual modeling in practice, *Data and Knowledge Engineering*, 58(3), 358–380.
2. OWL-S: Semantic Markup for Web Services, available at: http://www.ai.sri.com/daml/services/owl-s/1.2/overview/, Access at 12 Aug 2016.
3. J. Mending, K.B. Lassen, and U. Zdun, On the transformation of control flow between block-oriented and graph-oriented process modeling languages, *International Journal of Business Process Integration and Management*, Vol. 3, No. 2, pp.96-108, 2008.
4. M Rosemann, WMP van der Aalst. A Configurable Reference Modelling Language, *Information Systems*, 2007, Elsevier
5. Keqing He and Chong Wang.ISO/IEC 19763-5:2015, Information technology- Metamodel for interoperability – Part 5:Metamodel for process model registration,http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53761
6. Frederic Jouault, Freddy Allilaire, Jean Bezivin, and Ivan Kurtev, ATL: A model transformation tool, *Science of Computer Programming*, 72: 31-39, 2008.
7. Arthur H.M. ter Hofstede, Chun Ouyang, Marcello La Rosa, et al., APQL: A Process-Model Query Language, In: *AP-BPM 2013*, LNBIP 159, pp. 23-38, 2013.
8. Object Management Group, MOF model to text transformation language, V 1.0, available at: http://www.omg.org/spec/MOFM2T/About-MOFM2T/, 2018.
9. Chun Ouyang, Marlon Dumas, Wil M.P. Van der Aalst, Arthur H.M. Ter Hofstede, and Jan Mending, From Business Process Models to Process-Oriented Software Engineering, *ACM Transactions on Software Engineering and Methodology*, Vol.19, No.1, 2009.
10. La Rosa, Marcello, et al. "APROMORE: An advanced process model repository." *Expert Systems with Applications*, 38.6 (2011): 7029-7040.
11. Business Process Model and Notation (BPMN) Version 2.0, OMG, January, 2011. Available at: http://www.omg.org/spec/BPMN/2.0.
12. D.H. Akehurst, B.Bordbar, M.J. Evans, et al., SiTra: Simple Transformations in Java, In: *MoDELS 2006*, LNCS 4199, pp. 351-364, 2006.
13. Business Process Execution Language for Web Services. (BPEL 1.1), Availableat: http://xml.coverpages.org/BPELv11-May052003Final.pdf
14. Esther Guerra, Juan de Lara, Manuel Wimmer, et al., Automated verification of model transformations based on visual contracts, *Automated Software Engineering*, 20: 5-46, 2013.
15. G. Decker, H.overdick, M. Weske, Oryx: An open Modeling Platform for the BPM Community, In: proceeding of BPM 2008, PP. 382-385.
16. D.K.W. Chiu and H.F. Leung. Towards ubiquitous tourist service coordination and integration: a multi-agent and semantic web approach. In: Proc. of the 7 th Int'l Conf. on Electronic Commerce (ICEC'05): 574-581.
17. D.K.W. Chiu, S.C. Cheung, and H.F. Leung. A Multi-Agent Infrastructure for Mobile Workforce Management in a Service Oriented Enterprise. In System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on (pp. 85c-85c). IEEE.
18. BehzadBordbar Gareth Howells, Michael Evans, and et al., Model Transformation from OWL-S to BPEL via SiTra, In: *ECMDA-FA 2007*, LNCS 4530, pp. 43-58, 2007.
19. Injun Choi, HyunbaeJeong, Minseok Song, and Yong U. Ryu, IPM-EPDL: an XML-based executable process definition language, *Computer in Industry*, 56 (2005) 85-104.
20. B.F. van Dongen, M.H. Jansen-Vullers, H.M.W. Verbeek, W.M.P. van der Aalst, Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants, *Computer in Industry*, 58 (2007) 578-601.
21. JussiVanhatalo, Jana Koehler, and Frank Leymann (2006) 'Repository for business processes and arbitrary associated metadata', Proceedings of International Conference on Business Process Management, Springer, pp.25-31.
22. Thomas Theling, JorgZwicker, Peter Loos, and Dominik Vanderhaeghen (2005) 'An Architecture for Collaborative Scenarios Applying a Common BPMN- Repository', Proceedings of Distributed Applications and Interoperable Systems, Springer, pp. 169-180.
23. Jan Mendling and Jorg Ziemann, Transformation of BPEL Processes to EPCs, In: Proceedings of the 4th GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2005), pp. 41-53.
24. G. Keller, T. Teufel, SAP R/3 Process Oriented Implementation: Iterative Process Prototyping, Addison-Wesley. 1998.
25. Wang C, Luo Z, Zhang X, et al. An approach to business process registration for enterprise collaboration: using BPEL as an

example[J]. *International Journal of Business Process Integration and Management*, 2015, 7(3): 181-196.

26. Cheng J., Wang C., He K., Jia J., Liang P. (2012) Mappings from BPEL to PMR for Business Process Registration. In: Camarinha-Matos L.M., Xu L., Afsarmanesh H. (eds) Collaborative Networks in the Internet of Services. PRO-VE 2012. IFIP Advances in Information and Communication Technology, vol 380. Springer, Berlin, Heidelberg
27. ISO/IEC 19505-2, Information technology - Object Management Group Unified Modeling Language (OMG UML) - Part 2: Superstructure.
28. ISO 18629-1:2004, Industrial automation systems and integration - Process specification language - Part1: Overview and basic principles.
29. IDEF3 Process Description Capture Method Report, September 1995. Available at: http://www.idef.com/pdf/Idef3_fn.pdf.
30. Wen Zhu, Implementation of Process Reuse Technology based on Metamodel for Process Model Registration, Master Thesis, Wuhan University, 2011.
31. R. Khadka, B. Sapkota, Luis. F. Pires, et al., WSCDL to WSBPEL: A Case Study of ATL-based Transformation,In: 3rd International Workshop on Model Transformation with ATL(MtATL2011), CEUR Workshop Proceedings, ISSN 1613-0073, 2011.
32. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, November 2005. World Wide Web Consortium (2005), Available at: http://www.w3.org/TR/ ws-cdl-10/
33. Object Management Group, QVT Specification Version 1.1. http://www.omg.org/spec/QVT/1.1/, 2011.
34. Dimitrios S. Kolovos, Richard F. Paige and Fiona A.C. Polack, The Epsilon Transformation Language, In: ICMT'08. LNCS,vol. 5063, pp.46-60. Springer, Berlin, 2008.