Alignment and Change Propagation between Business Processes and Service-Oriented Architectures

Karim Dahman, François Charoy, Claude Godart

Alignment and Change Propagation between Business Processes and Service-Oriented Architectures

Karim Dahman, François Charoy, Claude Godart
Université de Lorraine - LORIA
BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France
{*dahman, charoy, godart*}*@loria.fr*

***Abstract*—Increasing the productivity of the Service-Oriented Software Engineering through a model-driven methodology needs to go beyond business services modeling and the automation of transforming models that adhere to the SOA style. Commonly, the business analysts focus on modeling business processes. Then, IT developers implement software architectures that are automatically generated from the process models. In this paper we look forward to maintaining the alignment between business services and their supporting IT assets when business settings evolve. We introduce an approach for an incremental synchronization between business process models and component configuration models that follow SOA architectural principles. We automate the change forward propagation by enforcing in-place translations of the updates on the models in order to preserve their consistency. Rather than executing the entire development process again, software architects can assess the necessity of propagating design decisions to the implementation level by adapting the deployed IT assets according to the business evolutions.**

***Index Terms*—Business Process, SOC, MDE, SOA Alignment**

I. Introduction

Maintaining the alignment between evolving business solutions and their supporting IT capabilities is essential for enterprise information systems [1]. Service-Oriented Computing [2] has emerged as an approach for modeling, building and managing software applications on the basis of Service-Oriented Architecture [2] (SOA). Software architectures that adhere to this style are implemented as loosely-coupled software components that are composed in configurations and linked by connectors. Those service-component realize business services at the implementation level by enacting process tasks. They expose their functionalities as business services, and consume other services in an uniform way [3].

Business-Driven Development experience shows that using Model-Driven Engineering [4] (MDE) enables to align software architectures with the business processes. During the initial development activities, IT developers can rapidly instantiate software architectures from business processes with an automated model transformation. However, this *Business-IT alignment* can disappear when business requirements evolve incrementally. Here, we want to enable the behavioural and architectural evolutions of business processes that imply structural adaptations. We support the adaptation of service-component implementations when the business processes are modified to maintain their alignment.

To achieve that goal, we propose an incremental model synchronization framework. Following the generative approach of our previous work [5], we enforce a conceptual mapping between a business process model and a service-component configuration model. We automate the change forward propagation to translate consistently an update on the process model into the service model when a transformation was previously executed. We base our work on the formal foundations of graph rewritings [6], [7] to preserve the model consistency. We have developed a proof-of-concepts prototype to validate our method of the consistency management between the models.

The paper is organized as follows. In Section II, we sustain the usage of two domain-specific languages for our MDE approach to bridge the gap between cross-organizational processes modeling and SOA development. Section III formally describes our framework and Section IV presents our incremental model synchronization principles. In Section V, we demonstrate a proof-of-concept implementation of those principles with a discussion on their theoretical and experimental evaluations. Finally, we situate our work with the related research in Section VI, and we present future possible extensions in Section VII.

II. Bridging the Gap between Processes and SOA

In this section, we present an example (inspired from [8]) of a *Supply Network* which illustrates some situations that occur when software architects have to propagate business process evolutions to the supporting service-component logic.

To model the business logic [3] of each partner in the network we use the Business Process Model and Notation [8] (BPMN) standard. It refers to a service orchestration [2] as a set of ***participants*** with ***processes*** in a ***collaboration***. BPMN combines graphical and textual annotations to describe business process models with cross-domain capability sharing that are decoupled from their supporting software architectures. The BPMN diagram ***s*** in Figure 1 shows business service interactions between the collaborating partners (*Supplier*, *Customer*, *Shipper* and *Invoicer*). It specifies the behavioural (***flows*** between ***tasks***) and architectural (***conversation links*** between ***participants***) views of the business partners. The information exchanges are modeled as ***conversations*** that capture service contracts and interactions. The dotted box in ***s***

describes the design decisions made by the business analysts for the *Ordering* service (between *Supplier* and *Customer*) with a send-receive interaction pattern [9]. It is captured by a single BPMN conversation. We consider that the core BPMN constructs subset, drawn in the legend of Figure 1, is sufficient to capture the most important service-oriented design aspects.

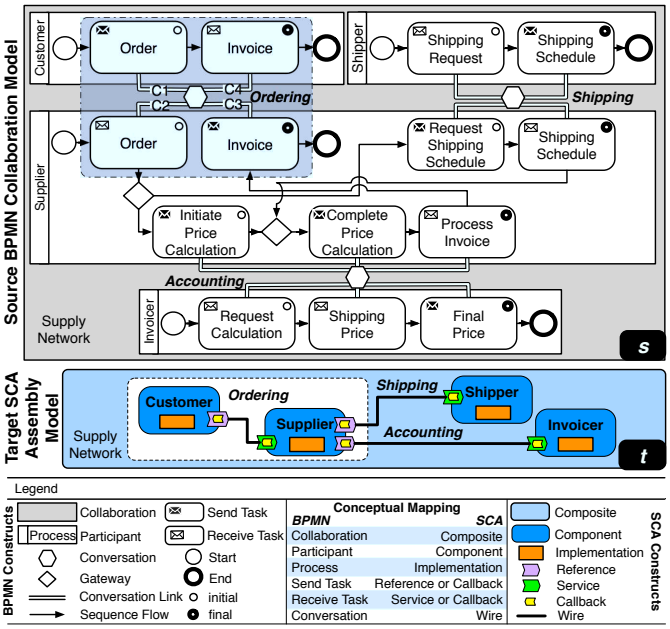

Fig. 1. BPMN-to-SCA model transformation example.

In [5], we provided a development scenario to transform a source BPMN collaboration model into a target service-component configuration model described using the Service-Component Architecture standard [10] (SCA). SCA provides a model for composing applications that embraces SOA principles. We refer to the program executions that implement the BPMN-to-SCA conceptual mapping (functional relation among BPMN and SCA constructs [11]) as model transformations. For example, ***s*** generates the model ***t*** of Figure 1. A simplified conceptual mapping is shown in the figure's legend.

In order to automate model transformations, we conceptually map BPMN collaborations to SCA composites. Likewise, the BPMN participants map into SCA components. Those composites modularize and compose service-enabled business functionalities in a manner that is decoupled from their implementation code. For example, ***t*** describes a composite (*Supply Network*) of four components (*Supplier*, *Customer*, *Shipper*, *Invoicer*). It specifies the behavioural (***flows*** between ***tasks***), their service dependencies and other related artifacts which specify how they are consumed and offered. Each component exposes ports, also called services. It requires other services by means of ***references***. It's configured to interact with the other components through ***wires***. When a component is implied in bidirectional interaction patterns, ***callbacks*** are defined for the services and/or references (see *Customer* and *Supplier* in the dashed box of ***t***). The callback of the consumer component (*Customer*) is used by the provider (*Supplier*).

The SCA wire is conceptually similar to the BPMN conversation construct. It represents an *activation logic* between two components and means an explicit message exchange. However, there is no mean of control flow between SCA services and references: each of them represents the direction of the first message exchanged between two components. Therefore, conversations between participants map[1] into wires. A BPMN task maps into SCA service or reference, depending when it sends the first or the last message of the collaborations. For this purpose, we add an explicit *index* for each task in each process by extending the BPMN standard notation with the two icons: *init* and *fin*. The icon *init* (resp. *fin*) indicates an ***initial*** (resp., a ***final***) task that sends or receives the first (resp. last) message in a service interaction pattern [9]. By introspecting process structures, an initial receive task is mapped to a service. Accordingly, a first send task maps to a reference. After the first receive task (resp., first service task), a process can contain a range of other send tasks (resp., receive tasks) that interact with the same participant. Thus, a final send task (resp., receive task) maps to a callback that is associated with the first mapped reference (resp., service).

The SCA composites can be deployed in a runtime engine that provide binding mechanisms and maintain the neutrality between component composition logic and their implementation code. Each component contains an implementation [10] of service orchestration process with a suited programming language. BPMN processes map into SCA implementations. In this sense, BPMN processes can be transformed[2] into other process execution languages, or even directly enacted with a BPMN-compliant workflow engine.

A. Maintaining alignment between Business Process and SOA

Now, we consider that business analysts need to change the *Supply Network* by separating the services concerns of *Ordering* and *Invoicing* as a best way to deal with their business requirements. They make a design decision to split the *Ordering* service (modeled in the *as-is* source model ***s*** of Figure 1). As depicted in (the *to-be* source model) ***s'*** of Figure 2, they update ***s*** with two separate conversations for send-after-receive and receive-after-send patterns [9].

Then, software architects have to propagate this source BPMN model change (so-called delta) to the previously generated target SCA model ***t*** of Figure 1. According to the BPMN-to-SCA conceptual mapping [11], they must adapt the (*as-is*) target model ***t*** to a (*to-be*) target model ***t'*** as depicted in Figure 2. The software architects have to add a wire (*Invoicing*) between a service and a reference, and delete two callback. Here, change forward propagation is challenging since similar BPMN constructs are transformed into different SCA constructs: task *Invoice* maps to a callback in ***t*** and to a service in ***t'*** (because of the multivalued[3]

[1]We mainly focus on the mapping of sequence and message flows, and put the transformation of data flows between into language-agnostic data structures (being generated) out of this article's scope.

[2]We assume well-formed and well-behaved business processes [9], [12].

[3]There maybe multiple SCA models for the same BPMN model [11].

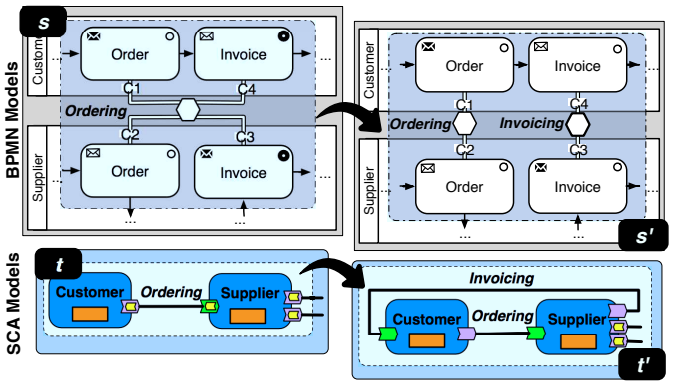

Fig. 2. BPMN and SCA model evolution use case.

mapping relations). Manual adaptations of the SCA model are potentially error-prone [9]. Especially, they can lead to misalignments if they do not follow a strict roadmap and a thorough delta analysis between *as-is* and *to-be* models [13]. Thus, enabling uncontrolled updates on $\boldsymbol{t}$ is problematic.

Executing a full transformation of $\boldsymbol{s}$' to get another model $\boldsymbol{t}_*$' and automatically adapting $\boldsymbol{t}$ to $\boldsymbol{t}_*$' is possible. We can operate an automated model difference (so-called *diff*), so that the difference and overlapping between the two model versions $\boldsymbol{t}$ and $\boldsymbol{t}_*$' can be extracted. Then, we can compute $\boldsymbol{t}$' by adapting $\boldsymbol{t}$ with the *diff*. However, a full transformation of $\boldsymbol{s}$ (when it evolves) is not suitable since any additional information in $\boldsymbol{t}$ is lost when applying the *diff*. The elements in $\boldsymbol{t}$ which are not covered by the overlapping cannot be derived automatically in $\boldsymbol{t}$' and are thus lost. Hence, a practical approach should not replace $\boldsymbol{t}$ by $\boldsymbol{t}_*$', but it has to reuse $\boldsymbol{t}$ in order to preserve extensions and refinements (crucial prerequisites to foster the MDE paradigm). Reusing previous SCA implementations for a new component composition logic and tracking the change impact at the architectural level becomes very difficult.

Moreover, when enforcing a *diff*-based model synchronization [14], the change roadmap computation effort (to obtain *to-be* models from *as-is* ones) is proportional to the size of the models and not to the size of the updates. Recomputing a full transformation even though only a small fraction of the source model has been modified becomes very costly, especially when a frequent synchronization is considered. In practice, when the business logic evolves with incremental and localized updates that do not radically modify the BPMN model, we have to operate complex computations to obtain SCA models. For this reasons we want to *incrementally* adapt them.

Existing general synchronization frameworks [15] and transformation languages [14] cannot work well for a functional and multivalued mapping [11], since they are intended for generic model transformations. First, they require users to explicitly write a synchronization code to deal with each update, and on each domain-specific language. Second, mapping complexity of each change is compounded with the decisions regarding information loss or gain related to different levels of heterogeneous synchronizations [14]. Thus, what sounds so straightforward in theory while using a model-driven approach, turns out to a very challenging endeavor in practice. To foreshadow a consistent maintenance of the *Business Process-SOA* alignment and change propagation automation, we introduce a framework which guarantees the *structural consistency* between our assorted domain-specific languages. We consider that this *incremental BPMN-to-SCA model synchronization* is a major use case for preserving the *Business-IT alignment*.

III. Model Synchronization Framework

This section formalizes our incremental model synchronization framework. We also introduce the requirements that have to be satisfied by a tool support (so-called *synchronizer* [6]). Our following explanations are based on Figure 3. It shows the relations between the models of Figure 2.

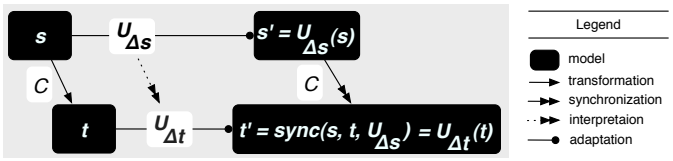

Fig. 3. Formalization of the incremental model synchronization.

The key difference between the *diff*-based and the incremental synchronization is in remembering previous transformation results. The former discards previous transformation results and recomputes new ones, while the latter updates them. It synchronizes two models by a change propagation. It preserves the information which is not covered by the transformation and minimizes the computational effort. Formally, consider that a BPMN-to-SCA model transformation is a partial function denoted by $trans : S \rightarrow T$ that takes a source BPMN collaboration model from a set of BPMN models S and produces a target model in a set of SCA models T. Executing $trans$ establishes a ***consistency relation*** between two models which is denoted by $\mathcal{C} \subseteq S \times T$. This relation is derived from the conceptual mapping between BPMN and SCA. For example, consider a BPMN model $\boldsymbol{s}$ in S and a SCA model $\boldsymbol{t}$ in T being transformed such $trans(\boldsymbol{s}) = \boldsymbol{t}$. Those models are ***consistent*** with respect to the binary relation as $(\boldsymbol{s},\boldsymbol{t}) \in \mathcal{C}$. Furthermore, given a source update $U_{\Delta\mathcal{S}}$ that alters $\boldsymbol{s}$ to $\boldsymbol{s}$' such it produces inconsistent models with $(\boldsymbol{s}',\boldsymbol{t}) \notin \mathcal{C}$, the *unidirectional incremental source-to-target synchronization framework* consists in computing a target update $U_{\Delta\mathcal{T}}$ from $U_{\Delta\mathcal{S}}$ such that the application of the both updates results in consistent models: $(\boldsymbol{s}',\boldsymbol{t}') \in \mathcal{C}$. It produces $\boldsymbol{t}$' by adapting $\boldsymbol{t}$ such as $U_{\Delta\mathcal{T}}(\boldsymbol{t}) = \boldsymbol{t}'$ and $U_{\Delta\mathcal{S}}(\boldsymbol{s}) = \boldsymbol{s}'$ and establishing the relation $\mathcal{C}$. We denote the ***incremental synchronization*** function by $sync : \mathcal{C} \times \Delta_{\mathcal{S}} \rightarrow T$. The updates on the target model are obtained by directly interpreting the source model updates. We denote this ***interpretation*** by a function $interp : \Delta_{\mathcal{S}} \rightarrow \mathcal{P}(\Delta_{\mathcal{T}})$ that interprets source changes in $\Delta_{\mathcal{S}}$ into a partition of target changes $\Delta_{\mathcal{T}}$.

To automate the incremental BPMN-to-SCA synchronization and to be compliant with our formalization, our synchronizer must implement a ***decidable*** algorithm. We consider that its execution (when synchronizing two models) has to

satisfy the following criteria: ***uniformity***, ***validity***, ***conformity***, ***stability***, ***autonomy*** and ***idempotence***. The *uniformity* and *validity* criteria require that it must produce consistent models. The *conformity* means that the resulting target models must conform to the SCA metamodel[4]. The *stability* means a non-destructive behavior on the SCA model when the BPMN model is not modified. The *autonomy* and the *idempotence* mean that it must returns the same proposed modifications on the target model when merging source models or their changes.

In practice, the synchronizer detects BPMN models changes. It translates them into SCA updates. When propagating those updates, it creates elements in the target if such elements do not exist. It modifies elements if such elements exist but have changed, and it deletes elements otherwise. These principles are explained below.

A. Abstracting Models to Graphs for Change Detection

In order to detect BPMN model evolutions and to adapt the SCA model, we relate models to *labelled nested typed rooted* graphs [7]. Those graphs provide an intuitive and general[5] formalism to represent BPMN and SCA models. We consider the BPMN and SCA models as collections of typed objects and links with some structural constraints defined by the BPMN and SCA metamodels. A node can represent any kind of model object. Edges are used to represent all kinds of associations between objects. Nesting in graphs implies a number of constraints for model instantiations that must be enforced when models evolve. For this purpose, we use graphs with distinguished containment edge type and syntactic constraints to make model conform to metamodels.

We assimilate each model to a tuple $(\mathcal{V}, \mathcal{E}, \mathcal{B}, \mathcal{Y}, \mathcal{I}, \mathcal{Z})$ that includes the functions ***label***:$\mathcal{V} \cup \mathcal{E} \rightarrow \mathcal{B}$, ***type***:$\mathcal{V} \cup \mathcal{E} \rightarrow \mathcal{Y}$, ***source, target***:$\mathcal{E} \rightarrow \mathcal{V}$, and ***index***:$\mathcal{V} \rightarrow \mathcal{I}$. The functions express the labeling, the nesting and the typing in a graph. The tuple contains disjoint sets of node identifiers $\mathcal{V}$, edge identifiers $\mathcal{E}$, node and edge labels $\mathcal{B}$, nodes and edge types $\mathcal{Y}$, indexes $\mathcal{I}$, and a structural correctness relation to express the nesting and associations constraints denoted by

$$\mathcal{Z} = \{(v, e, v', i) \in \mathcal{V} \times \mathcal{E} \times \mathcal{V} \times \mathcal{I} \mid index(v') = i \wedge source(e) = v \wedge target(e) = v' \wedge type(e) = \textbf{\textit{cons}}(type(v), type(v'))\}.$$

The function ***cons***:$\mathcal{Y}^2 \rightarrow \mathcal{Y}$ captures the language-specific syntax as a relation between model object types according to the metamodel. We use the relation $\mathcal{Z}$ to avoid the definition of an additional relation to capture the conformity relation between a model and its metamodel. For example, we write the equality $cons(Collaboration, Participant) = Contain$ to mean that BPMN collaborations contain participants.We write $cons(Wire, Reference) = Source$ and $cons(Wire, Service) = Target$ to express that SCA wires can be connected to references or services.

[4]Which offers the vocabulary for formulating reasonings on top of models.
[5]Graph transformations are proven in [16] for all category of graphs.

Given a BPMN model $\boldsymbol{s} = (\mathcal{V}_s, \mathcal{E}_s, \mathcal{B}_s, \mathcal{Y}_s, \mathcal{I}_s, \mathcal{Z}_s)$, and a SCA model $\boldsymbol{t} = (\mathcal{V}_t, \mathcal{E}_t, \mathcal{B}_t, \mathcal{Y}_t, \mathcal{I}_t, \mathcal{Z}_t)$, we express our multivalued BPMN-to-SCA conceptual mapping with a partial function $map : (\mathcal{Y}_s \times \mathcal{I}_s) \rightarrow \mathcal{Y}_t$ where $\mathcal{Y}_s$ denotes source types, $\mathcal{Y}_t$ denotes target types and $\mathcal{I}_s$ denotes model constructs indexes. We define our BPMN-to-SCA mapping relation as

$$\mathcal{CC} = \{(y, i, y') \in \mathcal{Y}_s \times \mathcal{I}_s \times \mathcal{Y}_t \mid y' \subseteq map(y, i)\}.$$

For example, *(Participant,∗,Component)* $\in \mathcal{CC}$ means that BPMN participants map into SCA components with their nesting constraints. Also, mapping BPMN initial send tasks to SCA references is given by *(SendTask,init,Reference)* $\in \mathcal{CC}$. Finally, we denote the ***consistency relation*** as a binary relation between source and target objects

$$\mathcal{C} = \{(o, o') \in (\mathcal{V}_s \times \mathcal{V}_t) \cup (\mathcal{E}_s \times \mathcal{E}_t) \mid \exists i \in \mathcal{I}_s : (type(o), i, type(o')) \subseteq \mathcal{CC}\}.$$

IV. Automating the incremental synchronization

In order to sustain automated synchronizations, we have built a synchronizer which is based on three principles: change detection, change impact analysis and change propagation. We present those principles in the following sections.

A. Change Detection as Primitive Operations on Graphs

Since we relate a model to a graph, an update on a model describes a modification on a graph structure. Model evolutions can be expressed by graph rewritings [16] and modeled by graph productions[6]. A production is defined through application rules which consist of in-place graph updates that are performed with a description of positive and/or negative patterns. Those conditional graph productions [17] are given through structural graph properties, while graph rewritings are usually defined through push-out constructions [16].

Compared to our previous work [13], here, we investigate heterogeneous incremental model synchronizations. Actually, we have shown that conditional graph productions can be expressed by graph update operations in order to guarantee the correctness of the model evolution. Now, we examine in more details how to express compound model updates with primitive operations on atomic constructs. This approach offers finer model consistency management.

TABLE I
PRIMITIVE UPDATE OPERATIONS AND THEIR APPLICATION RULES.

Operation	Precondition	Invariant	Postcondition
$create(\phi)$	$-\phi$	$-(\phi, *, *)$	$+\phi$
$destroy(\phi)$	$+\phi$	$-(\phi, *, *)$	$-\phi$
$update(\phi, \mu, \nu)$	$-(\phi, \mu, \nu)$	$+\phi$	$+(\phi, \mu, \nu)$
$undo(\phi, \mu, \nu)$	$+(\phi, \mu, \nu)$	$+\phi$	$-(\phi, \mu, \nu)$

Table I shows a complete and minimal set of four primitive update operations and their corresponding application rules. This set is complete because we can indicate all the structural modification in a model assimilated to a graph. It is also

[6]They are based on the concept of gluing graphs and graph morphisms [16]. They describe how to modify a right-hand graph to produce a related left-hand graph with forward transformations [7].

minimal because we can not replace operations with each other to express the same update. For example, assigning the type *Task* and the label *Order* to an existing object with an identifier θ is given by $update(\theta, type, Task)$, $update(\theta, label, Order)$. To create an edge with identifier ω and assign a target node θ to this edge, we use $create(\omega)$, $update(\omega, target, \theta)$. We also define the opposite operations for deleting an object from a graph and undoing a property of an object. The operation *update* can be considered as redo operation for the *undo*.

Our application rules include finite negative or positive application conditions that must be satisfied by each graph rewriting. Before applying an operation, it is necessary to verify *invariants* and verify that the *precondition* is fulfilled in the model that should be updated. Thereafter, the existence of positive or negative patterns has to be checked. If they are verified, *invariants* have to be checked and then updates can be applied and *postconditions* verified. For example, before applying the operation $add(\phi)$, we must avoid that an object with the same identifier ϕ already exists. The effect of this operation is that ϕ is added to the graph. Those rules allow to avoid structural conflicts by detecting contradictions between assertions. Table II presents the assertions used in Table I.

TABLE II
LIST OF ASSERTIONS AND THEIR CONTRADICTIONS.

Assertion	Notation	Contradicts
Object ϕ exists	$+\phi$	$-\phi$
Object ϕ has ν as a value for μ	$+(\phi, \mu, \nu)$	$-(\phi, \mu, \nu)$
Object ϕ has no properties	$-(\phi, *, *)$	$+(\phi, \mu, \nu)$

The assertions are well established in the software community as a formal way to specify the applications rules for conditional graph productions [18]. In order to describe contradicting primitive operations, they are attached to graph productions operation rather than to graphs – we consider that models are related by morphisms that are totally label-preserving, type-preserving [17] and root-preserving [7]. We distinguish between positive and negative assertions to express the presence or the absence of a structural property. The negative assertions are denoted by a minus sign and mean the opposite of the positive assertions. We use wild-card assertions to express transitive constraints. For example, $-(\phi, *, *)$ means that in order to assign properties to ϕ we must apply an *update* operation. Also, to apply an *undo* on a property μ with a value ν, $+(\phi, \mu, \nu)$ imposes that this property is already assigned to ϕ. Thus, we can not replace *update* with *undo*.

B. Change Detection as Compound Operation Sequences

The business analysts can edit a BPMN model in many places during a modeling session. The structural correctness checking of model updates is complex since it relates to the combinatorial explosion of the graph patterns [19]. We decompose the model updates into primitive operations on atomic constructs. We can record their changes on a model ***s*** as a finite sequence of primitive operations that result in a model ***s'*** $= U_{\Delta\mathcal{S}}(s)$ such as the update

$U_{\Delta\mathcal{S}}$: $\{\delta_i : S \rightarrow S \mid \delta_i$ *is a graph production* $\}_{1 \leq i \leq n} \in \Delta_{\mathcal{S}}$.

In order to reduce the checking complexity and to comply with the BPMN and SCA modeling tools, we define atomic combined graph update operations as well-formed sequences of primitive operations. Tables III gives a non-exhaustive list of combinations. Combining operations provides enhanced semantics to express much meaningful updates [17], [13]. We state that two primitive operations in a sequence are distinct, if they affect independent model objects [15], [20]. It means that they can commute in the sequence without changing the result. The syntactic correctness of a sequence can be checked if the assertions imposed by an operation in the sequence do not contradict assertions of the earlier operations as given in Table II. An update sequence $\{\delta_i\}_{1 \leq i \leq n}$ is *well-formed*[7]

$$\textbf{\textit{if}}\ \forall \alpha_{pi} \in (\delta_k.pre \cup \delta_k.inv) \mid 2 \leq k \leq n \wedge \forall \alpha_{ip} \in (\delta_l.inv \cup \delta_l.pos) \mid l < k :\ \alpha_{pi}\ \textbf{\textit{does not contradict}}\ \alpha_{ip}.$$

This definition ensures that the syntactic correctness is also maintained for a compound update. For example, the sequence $\{update(\theta, label, Order), destroy(\theta)\}$ is ill-formed because assertion $-(\theta, *, *)$ in the invariants of $destroy(\theta)$ contradicts the assertion $+(\theta, label, Order)$ in the postcondition of $update(\theta, label, Order)$. Actually, we have to remove all object properties before to destroy it.

Assuming that a modeling session starts with a structurally correct model which conforms to his metamodel, the resulting graph of an update should be a structurally correct model that conforms to the same metamodel. In order to check the conformity of the resulting model after applying a compound update, we verify operation sequences towards the structural correctness relation (i.e., $\mathcal{Z}$ in the previous section). For example, consider that adding a receive task node to the model ***s*** in Figure 1 is expressed with *addNode(7, Invoice, ReceiveTask, 3, fin)*. This addition preserves the BPMN model structural correctness since by construction the operation *addNode(v,b,y,v',i)* in Table III verifies that *(v,e,v',i)*$\in \mathcal{Z}_s$, where *e* is the edge inserted for nesting *v* in *v*.

C. Change Impact Analysis as Propagating Updates

In the previous section, we have defined the necessary properties to guarantee correct update applications on each of the BPMN and SCA models. In this section, we introduce the foundations for our consistency management approach. Namely, we propose a method for a change impact analysis which ensures that compound operations are propagated form BPMN to SCA models in a consistent manner.

An update operation $\delta \in U_{\Delta\mathcal{S}}$ on the source model is propagated, if it forces a synchronization on the target model [14]. It has to establish the consistency relation between the source model and the target model such that it preserves $(U_{\Delta\mathcal{S}}(\boldsymbol{s}), U_{\Delta\mathcal{T}}(\boldsymbol{t})) \in \mathcal{C}$. Therefore, our synchronizer produces update operations $U_{\Delta\mathcal{T}}$ on the target model such that $U_{\Delta\mathcal{T}} = interp(\delta)$. Otherwise, δ is *non-propagating*. A simplified algorithm that provides the interpretation function $interp$ is listed in Algorithm 1.

[7] ***pre***, ***post*** and ***inv*** represent the assertion sets shown in Table III.

TABLE III
COMBINATIONS OF PRIMITIVE UPDATE OPERATIONS.

Compound Operation	Description	Primitive Operations
addNode(v,b,y,v',i)	Adding a node *v* of type *y* nested in node *v'* with a label *b* and an index *i*	*add(v), update(v,label,b), update(v,type,y), update(v,index,i), add(e), update(e,type,Contain), update(e,source,v'), update(e,target,v)*
insertEdge(e,b,y,v,v')	Inserting an edge *e* with a label *b* and a type *y* between nodes *v* and *v'*	*add(e), update(e,label,b), update(e,type,y), update(e,source,v), update(e,target,v')*
dropNode(v,b,y,v',i)	Deleting an existing node *v* from node *v'*	*undo(e,type,Contain), undo(e,source,v'), undo(e,target,v), undo(v,index,i), undo(v,type,y), undo(v,label,b), destroy(e), destroy(v)*
setIndex(v,i,i')	Updating the index of the node *v* from *i* to *i'*	*undo(v,index,i), update(v,index,i')*
setSource(e,v,v')	Updating the source of the edge *e* from *v* to *v'*	*undo(e,source,v), update(e,source,v')*

The algorithm is written with inference rules of the form: **if** *premises* / **then** *conclusions*. The *premises* refer to operations and conditions on the source BPMN model, and *conclusions* refers to the propagating operations on the target model. As stated in Section III, our incremental synchronizer takes as parameters an update on the source model $U_{\Delta\mathcal{S}} \in \Delta_{\mathcal{S}}$ ($U_{\Delta\mathcal{S}}(\boldsymbol{s}) = \boldsymbol{s}'$) and two consistent models such ***(s, t)*** $\in \mathcal{C}$. Rather than executing the entire novel transformation, it enforces an in-place synchronization by interpreting $U_{\Delta\mathcal{S}}$ into a target update $U_{\Delta\mathcal{T}} \in \Delta_{\mathcal{T}}$ ($\boldsymbol{t}' = U_{\Delta\mathcal{T}}(t)$) to get $\boldsymbol{t}'$ consistent with ***s'*** such ***(s', t')*** $\in \mathcal{C}$ holds. It interprets $U_{\Delta\mathcal{S}}$ into $U_{\Delta\mathcal{T}}$ and then propagates them to ***t***.

Algorithm 1: Algorithm of the interpretation function.

in : $U_{\Delta\mathcal{S}}$ **out :** $U_{\Delta\mathcal{T}}$

foreach $\delta \in U_{\Delta\mathcal{S}}$ **do** `/* Initially` $U_{\Delta\mathcal{T}} = \emptyset$ `*/`

1 **if** δ=***addNode(***v, b, y, v', i***)*** $\wedge\ (v', v'') \in \mathcal{C} \wedge (y, i, y') \in \mathcal{CC}$
then $(v, v''') \in \mathcal{C} \wedge$ ***addNode(***v''', b, y', v'', i***)*** $\in U_{\Delta\mathcal{T}}$

else $\mathcal{C} \leftarrow \mathcal{C} \cup (v, \emptyset)$ `/* δ is non-propagating */`

2 **if** δ=***insertEdge(***e, b, y, v, v'***)*** $\wedge (v, v''), (v', v''') \in \mathcal{C} \wedge$
$(y, *, y') \in \mathcal{CC}$
then ***(e,e')*** $\in \mathcal{C} \wedge$ ***insertEdge(***e', b, y', v'', v'''***)*** $\in U_{\Delta\mathcal{T}}$

else $\mathcal{C} \leftarrow \mathcal{C} \cup (e, \emptyset)$ `/* δ is non-propagating */`

3 **if** δ=***setIndex(***v, i, i'***)*** $\wedge\ (v, v'), (v, v'') \in \mathcal{C} \wedge$
$(y, i, y'), (y, i', y'') \in \mathcal{CC}$
then ***dropNode(***v, b, y, v', i***), addNode(***v', b, y'', v'', i'***)*** $\in$
$U_{\Delta\mathcal{T}}$

4 **if** δ=***setSource(***e, v, v'***)*** $\wedge\ (e, \emptyset), (v', v''), (v''', v'''') \in \mathcal{C} \wedge$
$(y, *, y') \in \mathcal{CC}$
then $(e, e') \in \mathcal{C} \wedge$ ***insertEdge(***e', b, y', v'', v''''***)*** $\in U_{\Delta\mathcal{T}}$

In order to illustrate our interpretation algorithm, consider that the BPMN model ***s*** is altered to ***s'*** (i.e., the case study of Figure 2) with the well-formed update sequence for ***s'*** in the Figure 4. The *setIndex (7, fin, init, 3)* is interpreted to *dropNode(5, 4), addNode(5,Invoice,Service,2,init)* for ***t'*** by the rule in line **4**. It supposes that a propagating operation *addNode(7, Invoice, ReceiveTask, 3, fin)* on ***s*** was already translated to *addNode(5, Invoice, ReferenceCallback, 4, fin)* on ***t*** such $\mathcal{C}$ contains *(7,5)*. The rule in the line **5** translates *setSource (23, 20, 25)* into a propagating insertion of the edge with a label *C3* in ***t'*** and updates $\mathcal{C}$ with *(23,14)*.

Note that the consistency relation $\mathcal{C}$ serves as a working memory for tracking the interpretation of the update operations and transformation history-awareness. It keeps track of objects and links of the source model and their counterparts in the target model. The synchronizer locates the corresponding constructs in the transformation working memory that are affected by the source update. With the correspondence identified in the mapping relation $\mathcal{CC}$, it computes in-place operations on the target with values from the source. Due to our partial mapping, some updates on the source model – which involve BPMN types without corresponding SCA types – are not propagated to the target model. For example, the BPMN constructs such as gateways have no counterparts in the SCA space because they are not relevant for the SCA configuration topology. Thus, when an operation is non-propagating, the working memory $\mathcal{C}$ is updated with a tuple *(*,∅)* to mean that the BPMN objects have no counterparts in SCA.

Operation sequence for s'	C	Operation sequence for t'
setIndex (7, fin, init)	**(7,5)**	***dropNode (5, Invoice, Service, 4, fin)*** ***addNode(5, Invoice, Service, 2, init)***
setIndex (16, fin, init)	(16,9)	dropNode (9, Invoice, Reference, 8, fin) addNode (9, Invoice, Reference, 6, init)
addNode (25, Invoicing, Conversation, 1, 4)	(25,13)	addNode (13, Invoicing, Wire, 1, 4)
setSource (23, 20, 25)	***(23,14)***	***insertEdge (14, C3, WireSource, 13, 9)***
setSource (24, 20, 25)	(24,15)	insertEdge (15, C4, WireTarget, 13, 5)

Fig. 4. Update operations and consistency relations of Fig 2.

D. Change Propagation as Interpreting Updates

Finally, as depicted in Figure 3, our incremental synchronizer computes the traget model ***t'***$= sync(s, t, U_{\Delta\mathcal{S}})$ such as the updates $U_{\Delta\mathcal{S}}$ and $U_{\Delta\mathcal{T}}$ produce consistent models. It establishes the consistency relation ($U_{\Delta\mathcal{S}}(\boldsymbol{s})$,$\boldsymbol{t}'$) $\in \mathcal{C}$ by propagating the $U_{\Delta\mathcal{T}}$ to the target model ***t***. It adapt the SCA model ***t*** by computing ***t'***$= U_{\Delta\mathcal{T}}(\boldsymbol{t})$ rather than executing the full transformation of the BPMN model. We see that the benefits of such a MDE approach include generating a roadmap, namely the delta $U_{\Delta\mathcal{T}}$, to adapt the component configurations when business settings evolve. In practice, the SOA development stakeholders can rapidly assess the necessity to propagate adaptation of the SCA models into the application code by making different updates on the BPMN models. To help them to make rational decisions, we propose a tool to simulate and to evaluate this change impact of the different business evolutions that can be modeled in the business process models. This tool support is presented in the following section.

V. TOOL SUPPORT IMPLEMENTATION AND EVALUATION

In this section, we describe a proof-of-concept implementation of our principles (change detection, change impact analysis and change propagation) for a tool that sustains automated synchronizations. We also provide a theoretical and experimental evaluation of our synchronizer.

We have implemented our synchronizer as an extension to a BPMN Editor [21] to demonstrate that it can be integrated business process modeling suites. The editor is used to model a

source BPMN and model. We have developed a filter that logs the primitive BPMN editing operation sequences. After applying our interpretation algorithm, we obtain a graph editing operation sequence that manipulates SCA constructs. We apply this sequence to another graph to produce the SCA model. We have implemented our interpretation algorithm in the *Drools* [22] rules engine. Due to this implementation we could meet the synchronization execution criteria presented in Section III. The interpretation function is based on the *decidability* and *confluence* offered by the *Rete* algorithm [22] implemented in Drools. The *uniformity* and *validity* criteria are verified by construction since the $interp$ algorithm produces consistent models according to the conceptual mapping. We support *conformity* through the (primitive and combined) update operations that we have defined. We support the requirement of *stability* since we prohibit the application of the same update operations with specific conditions. We satisfy *autonomy* and *idempotence* since we assimilate model to graphs and their updates to conditional graph productions.

Our synchronizer behaves like a model transformation engine. Consider that two empty BPMN and SCA models are consistent. Building the BPMN model means applying BPMN updates that are contained in a non empty operation sequence. Synchronizing the two empty models with the BPMN update sequence produces the SCA update sequence which generates the SCA model. However, taking into account a source update sequence $U_{\Delta\mathcal{S}}$ on the source BPMN model that gives s', the effort to interpret those updates, $O(size(U_{\Delta\mathcal{S}}))$, should be lower than computing a full transformation of the altered source model, $O(size(s'))$. It results in a reasonable decoupling from the source model size when $size(U_{\Delta\mathcal{S}}) \leq size(s')$. Thus, when the source update is small, it usually corresponds to a small target update, and the performance improvement to interpret the update operations is expected to be high.

In order to experiment our incremental synchronizer, we use the size of update operations as a metric to quantify the size of models and their changes. The size of a model (i.e., cardinal of objects and links) can be quantified by making a difference between the number of ***create*** operations and ***destroy*** operations in the sequence which represents that model. The size of a change (i.e., the number updates) is similarly calculated from the sequence which represents that change. This metric is independent from the experimental environment considerations. Our experimental evaluations consist in generating random update sequences for 1000 BPMN models and 1000 BPMN changes. In this way, the size $size(U_{\Delta_\mathcal{S}})$ of the generated updates is independent from the size $size(s)$ of the generated models. The generated models do not contain meaningful business logic, however they conform the BPMN metamodel. The operation sequences are randomized to have different sizes in such a way they equally likely overlap with the BPMN-SCA mapping. Then, we pass the update sequences representing the BPMN models (i.e., s in Figure 3) into our synchronizer to obtain the update sequences representing the SCA models (i.e., t in Figure 3). Again, we enforce interpretations of the update sequences representing the BPMN changes (i.e., $U_{\Delta_\mathcal{S}}$ in Figure 3). This results in the update sequences representing the SCA changes (i.e., $U_{\Delta_\mathcal{T}}$ in Figure 3). We can sum the update sequences of $size(s)$ and $size(U_{\Delta_\mathcal{S}})$ to obtain $size(s')$. Summing $size(t)$ and $size(U_{\Delta_\mathcal{T}})$ gives $size(t')$.

The Figure 5 plots the distribution of BPMN models size $size(s')$ (blue cloud) and BPMN change size $size(U_{\Delta_\mathcal{S}})$ (red cloud) against the fraction $\frac{size(U_{\Delta_\mathcal{S}})}{size(s')}$. It also plots the distribution of SCA models sizes $size(t')$ (yellow cloud) and SCA change sizes $size(U_{\Delta_\mathcal{T}})$ (green cloud) against the fraction $\frac{size(U_{\Delta_\mathcal{T}})}{size(t')}$. This figure gives two interesting observations. First, it confirms that our synchronizer implements a partial function due to the BPMN-SCA mapping, since $size(t') \leq size(s')$ (yellow and blue clouds) and $size(U_{\Delta_\mathcal{T}}) \leq size(U_{\Delta_\mathcal{S}})$ (green and red clouds). Second, it illustrates the efficiency frame of the incremental synchronization. Actually, when $size(U_{\Delta_\mathcal{S}}) \lessapprox size(s')$ (i.e., the left side of the black lane) it is less complex to interpret the BPMN updates (red cloud) than the BPMN models (blue cloud). Otherwise, the size of the SCA updates (green cloud) becomes greater than the size of SCA models (yellow cloud). This empirical observations confirm the expected behavior of our synchronizer. Also, an unexpected behavior arises in the practical experiments. It concerns an almost linearity between the factors $\frac{size(s')}{size(s)}$ and $\frac{size(t')}{size(t)}$. Since the generated updates are not correlated to the models (i.e., s' are not proportional to s), then this supports that our synchronization behaves like a transformation.

Evidently, there are some limitations inherent to our operation-based method when $size(U_{\Delta_\mathcal{S}}) > size(s')$. For example, when adding two BPMN elements and deleting one of them it is less interesting to translate three operations than transforming the model (i.e., by neglecting the *diff* effort). To address this scalability issue we can reduce the update sequences by eliminating the redundancy and enforcing distinct operations. For example, an addition of a node followed by an update of node's attributes can be resolved to a single addition with the appropriate attributes. Also, opposite operations can be discarded. This *normalization* [18] algorithm is decoupled from the synchronizer since it is integrated with the BPMN editing operations filter. We refer the reader to [6], [16] for further discussions on the scalability concerns.

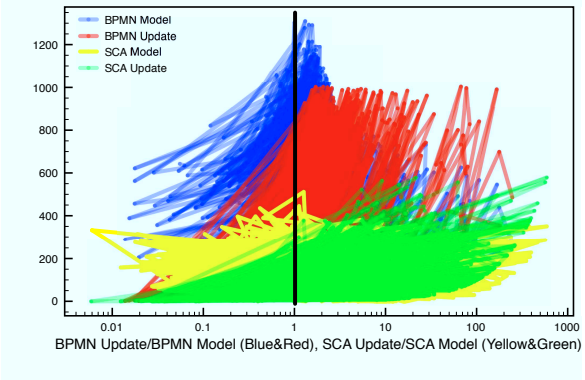

Fig. 5. Incremental synchronisation efficiency.

VI. Related Work

Building cross-organizational business systems with an SOA [2] that leverage the service-enabled process management best practices [3] implies a range of technical and methodological issues. First of all, the alignment between business settings and application functions has to be achieved with a Service Dominant Logic [1]: the Service-Oriented Software Engineering. This requirement should drive the production of the business services to achieve desired business processes. Second, business processes evolution is an aspect which affects the software life-cycle. There are many ways to change software artifacts. They range from the requirements evolution of the business processes to the SOA and the source code. Several frameworks have emerged to help organizations to charter successful route to Service-Oriented Computing [2]. The Model-Driven Architecture provides a mean for using UML-based transformations to drive the Service-Oriented Software Engineering [4]. However, as UML is neither fully service-oriented, nor business process-centered, it needs to address the gap between the emerging standard notations and the lower-level standards. A methodology suggested in [23] advocates the integration of the business values perspective to Service-Oriented Computing. It defines a methodology for the business process space, but unfortunately fails to provide a practical development scenario. Also, some attempts [24] provide tools to transform BPMN to SCA models. Unfortunately, they do not consider collaborative processes transformations and do not provide a mean for incremental model synchronizations.

To our knowledge, there is no tools providing automated incremental synchronization and change impact analysis framework with an advanced BPMN-to-SCA mapping. Only the *general model synchronization* frameworks [15] were used for the consistency management among heterogeneous models. Those frameworks give a general representation of the model updates, but rely on users to write code to handle each type of the modification in each metamodel [13]. Furthermore, QVT standard [25] encourages to use external model synchronizer. In our case, the synchronizer extracts informations automatically from previous model transformations and does not require users to write more synchronization code. Graph transformation theory allows us to compute conflicts and dependencies of transformation by relying on the idea of critical pair of operations (i.e., on two assorted models) analysis. Research conducted in the area of conditional graph rewriting [6], [7] has mainly been driven by theoretical computer scientists who put an emphasis on developing sound theory [16]. However, change propagation across heterogeneous software models remains an open problem [15]. In [17], the operation-based graph productions were used to express refactoring software transformations and their evolution. The usage of the assertions facilitates the detection of the syntactic merge conflicts when transforming the graphs. The graph transformations are executed in a uniform and scalable way. However, this technique is mainly focused on homogeneous model transformation, i.e., where the source and target models conform to the same metamodel. In our work, we perform a more thorough investigation on the heterogeneous incremental model synchronization.

VII. Conclusion and Future Work

Providing sound and effective ways to maintain the alignment between the business logic and its supporting software architectural logic is a challenge. It has become an important challenge today where services are pervasive and where BPM is mainstream in large companies strategy. In this paper, we have tackled this issue from the perspective of the relation between business process modeling and evolution and the corresponding Service Component Architecture design. Our approach not only allows to derive a component architecture from a business process collaboration, but, as demonstrated in this paper, allows to incrementally maintain this relation even in case of business evolutions. We have shown that a model-driven approach can be used to derive an abstract service architecture from a business process. It also serves to maintain their alignment when the process model is incrementally updated. By deriving changes that have to be applied to the architecture, we avoid to recreate an entire transformation. Our framework is integrated to a change impact simulation and analysis tool which is based on graph rewriting technique to avoid the inconsistencies caused by change propagations. Finally, we propose metrics for measuring the impact of business process evolutions on SOA maintenance and demonstrating model transformation/synchronization efficiency which ensures knowledgeable change governance. The scope of our contribution is limited so far to the Service-Oriented Software Engineering that uses BPMN and SCA languages. However, it can be generalized to other domain-specific languages and further domain modeling.

This work raises important questions. For example, if the behavioural properties (deadlock, controllability [9], etc) on a business process can be verified at design-time, are they still valid for the component configuration at the execution-time. Likewise, how IT developers can adapt their assets without losing design decisions and diverging from the business settings. We are also thinking about using it to consider the cloud as the runtime framework where SCA is used for integrating composite applications with the service delivery platform.

References

[1] H.-M. Chen, R. Kazman, and O. Perry, "From software architecture analysis to service engineering: An empirical study of methodology development for enterprise soa implementation," *TSC*, vol. 3, pp. 145–160, 10.

[2] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research chall." *Computer*, vol. 40, pp. 38–45, 07.

[3] M. Dumas and T. Kohlborn, "Service-enabled process management," in *Handbook on BPM 1*, ser. IHIS. Springer B., 10, pp. 441–460.

[4] D. Lopes, S. Hammoudi, J. Bézivin, and F. Jouault, "Generating transformation definition from mapping specification: Application to web service platform," in *CAiSE*, vol. 3520. Sprin. Ber., 05, pp. 183–192.

[5] K. Dahman, F. Charoy, and C. Godart, "Generation of Component Based Architecture from Business Processes: MDE for SOA," in *ECOWS*, Ayia Napa, 10, pp. 155–162.

[6] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *SoSyM*, vol. 8, pp. 21–43, 09.
[7] E. Biermann, C. Ermel, and G. Taentzer, "Precise semantics of emf model transformations by graph transformation," in *MoDELS*. Springer B., 08, pp. 53–67.
[8] *Business Process Model and Notation 2.0, Beta 1*, OMG, May 09.
[9] W. van der Aalst, A. Mooij, C. Stahl, and K. Wolf, "Service Interaction: Patterns, Formalization, and Analysis," in *WS-FM*. Springer B., 09, vol. 5569, pp. 42–88.
[10] *SCA Assembly Model Specification 1.1*, Open SOA, Mar. 09.
[11] K. Dahman, F. Charoy, and C. Godart, "From business process to component architecture: Engineering business to it alignment," in *EDOW*, Helsinki, 11, pp. 269–274.
[12] F. Puhlmann and M. Weske, "M.: Investigations on soundness regarding lazy activities," in *BPM*. Sprin. Ver., 06, pp. 145–160.
[13] K. Dahman, F. Charoy, and C. Godart, "Towards consistency management for a business-driven development of SOA," in *EDOC*, Helsinki, 11, pp. 267–275.
[14] M. Antkiewicz and K. Czarnecki, "Design space of heterogeneous synchronization," in *GRRSE II*. Springer B., 08, pp. 3–46.
[15] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *ASE*. New York, NY, USA: ACM, 07, pp. 164–173.
[16] Z. Diskin, "Algebraic Models for Bidirectional Model Synchronization," in *MoDELS*. Springer B., 08, pp. 21–36.
[17] T. Mens, "Conditional graph rewriting as a domain-independent formalism for software evolution," in *AGTIVE*, 00, pp. 357–359.
[18] ——, "Transformational Software Evolution by Assertions," in *CSMR*, Lisbon, 01.
[19] G. Taentzer and A. Rensink, "Ensuring structural constraints in graph-based models with type inheritance," in *FASE*. Springer B., 05, pp. 64–79.
[20] A. Habel and B. Hoffmann, "Parallel independence in hierarchical graph transformation," in *Graph Transformations*, ser. LNCS. Springer B., 04, pp. 207–210.
[21] [sourceforge.net/projects/bpmn/], "Yaoqiang bpmn editor," (Dec. 12).
[22] [www.jboss.org/drools/], "Drools (jboss rules)," (Dec. 12).
[23] C. Huemer, P. Liegl, R. Schuster, H. Werthner, and M. Zapletal, "Inter-organizational systems: From business values over business processes to deployment," in *DEST*, 08, pp. 294 –299.
[24] [www.eclipse.org/soa/], "Soa platform project," (Apr. 13).
[25] OMG, *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, Object Management Group Std., January.