Towards Consistency Management for a Business-Driven Development of SOA

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

***Abstract*—The usage of the Service Oriented Architecture (SOA) along with the Business Process Management has emerged as a valuable solution for the complex (business process driven) system engineering. With a Model Driven Engineering where the business process models drive the supporting service component architectures, less effort is gone into the Business/IT alignment during the initial development activities, and the IT developers can rapidly proceed with the SOA implementation. However, the difference between the design principles of the emerging domain-specific languages imposes serious challenges in the following re-design phases. Moreover, enabling evolutions on the business process models while keeping them synchronized with the underlying software architecture models is of high relevance to the key elements of any Business Driven Development (BDD). Given a business process update, this paper introduces an incremental model transformation approach that propagates this update to the related service component configurations. It, therefore, supports the change propagation among heterogenous domain-specific languages, e.g., the BPMN and the SCA. As a major contribution, our approach makes model transformation more tractable to reconfigure system architecture without disrupting its structural consistency. We propose a synchronizer that provides the BPMN-to-SCA model synchronization with the help of the conditional graph rewriting.**

***Index Terms*—Incremental Model Transformation, Structural Consistency Management, Business-IT Alignment**

I. Introduction

The Business Driven Development (**BDD**) refers to the discipline of building the **business processes** of an enterprise or between several organizations. It is used to develop the software solutions that directly satisfy the new business goals (or the requirements of the existing systems), and to provide a path from the business settings to the application code. During the BDD the business processes are mapped into the IT level, where they describe the configuration of the software applications that are organized in a Service-Oriented Architecture (**SOA** [1]). Moreover, with the SOA principles, the software applications are implemented as components with remotely published interfaces in loosely-coupled architectures. Generally, the SOA development extends the Component Based Development (**CBD**) paradigms [2], [3] that offers the means to define platform-independent architectures with the Service Dominant Logic [4]. In many cases, the experience shows that the initial business requirements vision is lost when the software architecture is instantiated [5]. One reason is the conceptual disparity between the BDD and the CBD disciplines and their domain-specific languages [6].

In a previous work [7], we have already introduced a Model Driven Engineering (**MDE** [8]) approach that automates the generation of (target) canonical **software architecture models** that are organized in a SOA from **business process models**. It is undeniable, however, that the flexibility to adapt a (source) business process model in order to respond to evolving business needs and automatically propagate its evolution to the (target) software architecture model is the most relevant trend in the success of this development method. What sounds so straightforward and easy in theory with the model transformation techniques turns out to a very challenging endeavor in practice. Even, if the used metamodels assume the same SOA concepts for building the software applications that provide and consume business meaningful services [2], however, not all source metamodel constructs correspond to target metamodel constructs due to the difference between their abstraction levels. The existing *general model synchronization frameworks* and *transformation languages* [9] cannot work well here. First, they require to manually write the synchronization code to deal with each update kind on each of the assorted models. Second, the complexity of the combinatorial change mapping is inherently compounded with decisions regarding the potential information loss or gain related to the different levels of the model's expressiveness.

When a source business process model evolves and a transformation has previously generated a corresponding technology-neutral SOA model, it is necessary to keep the models synchronized to enable rapid iteration of process redesigns and improvements of the underlying SOA implementations. This paper presents a **unidirectional model incremental transformation** approach. Its central contribution is the definition and the realization of an automatic synchronizer for managing and re-establishing the **structural consistency** of those heterogeneous source and target models. From a design-time perspective, an evolution describes the update on model's internal structures that can be assimilated to a graph. We express the model evolutions and their transformation with the help of the *conditional graph rewriting techniques* [10], [11]. We consider that the application of an update corresponds to the execution of a compound sequence of *primitive graph productions* [12] such as vertex or edge additions, removals, relocations, and attribute value changes. Our **synchronization algorithm**, that we implemented in a business process editor, works by translating the consistent source updates into con-

sistent updates onto the target software architecture model. If the update seems to be appropriate, then the IT developers can synthesize the platform-specific SOA implementations.

The remainder is structured as follows. In Section II, we introduce the languages that we use to describe the business process and the software architecture models with a case study. Then, we present the requirements of our incremental model transformation approach, and explain the consistency management for our MDE in Section III. This section gives also the details of our synchronization algorithm and its implementation. Finally, we situate our approach with the related research in Section IV, and conclude the paper in Section V.

II. Design-time Model Transformation

Typically, the Business Process Modeling (**BPM**) combines graphical and textual annotations to specify complex process-enabled systems that are decoupled from their supporting SOA architectures [6]. Several languages can be used to specify the business service interaction models at different levels of abstraction. The Business Process Modeling Notation (**BPMN** [13]) allows to specify systems that support cross-domain sharing capability as the lingua franca of the BPM. In particular, the **BPMN collaboration models** express and put in logical relation the messages exchanged between the internal processes of complex service-enabled business networks. They relate to coordinate an explicit exchange of information through (automated) business service interactions under the control of a single endpoint, called *service orchestration* [1]. During the BDD those business process models are mapped to the implementation models with a lower abstraction level, where they stand as a *business service interaction specifications* organized in a SOA. The SOA is an architectural style and a design principle for building software applications which promote loose coupling between the software components. It is often presented as an enabler of the Business Process Management because it enables the business process optimization along with flexible IT architectures [6]. The SOA has introduced a wide range of technologies and languages.

With the recent Service Component Architecture (**SCA** [14]) specifications, the CBD becomes a prominent and mainstream paradigm for implementing, composing and deploying of flexible software architecture models. The SCA defines an Architecture Description Language that can be used across many vendor environments. It can logically modularize and compose the service-enabled business functions in a manner that is decoupled from their implementations. With a graphical notation support, it describes composite assemblies of service-components, their connections and other related artifacts which specify how they are externally consumed and/or offered by other components. The so-called **SCA assembly models** specify a path from the business requirements and the business service interaction specifications to the executable processes and their implementations. At the execution-time, the assembly models are instantiated for the SCA runtime environments and benefit from those frameworks for the deployment and the monitoring. In [7], we have presented a MDE approach for an automatic transformation of a **(target) canonical SCA assembly model** from a **(source) BPMN collaboration model**, and specified relations among the BPMN and the SCA metamodels as **conceptual mappings**. However, the incremental BPMN-to-SCA tranformation remains an open question. We refer to the program executions that implement these conceptual mappings as **transformations**. Note that the usage of the BPMN along with the SCA is not only intended for modeling the process-enabled logics of the business services, but also for specifying the architectural settings of the cross-organizational business service that drive the SOA implementations. Section IV provides a detailed discussion on this choice.

A. Case Study

To study the issues of an incremental model transformation, we have chosen a motivating example that represents a *dealing network* [13]. In this service-enabled business network, a *supplier* participant provides an *order status* business service to a *customer* partner. This service is used when the customer has beforehand ordered goods, and thus it stands for their shipping. The two partners play both consumer and provider roles in this business service. Depending on the design choices, this service network can be modeled in different ways. For example, the Figure 1 shows the high-level diagrams of two source BPMN collaboration models, e.g., s' and s'', and their transformations into target SCA assembly models, e.g., t' and t''. Each element on the target model is labeled with the label's first letters of the source model from which it is generated.

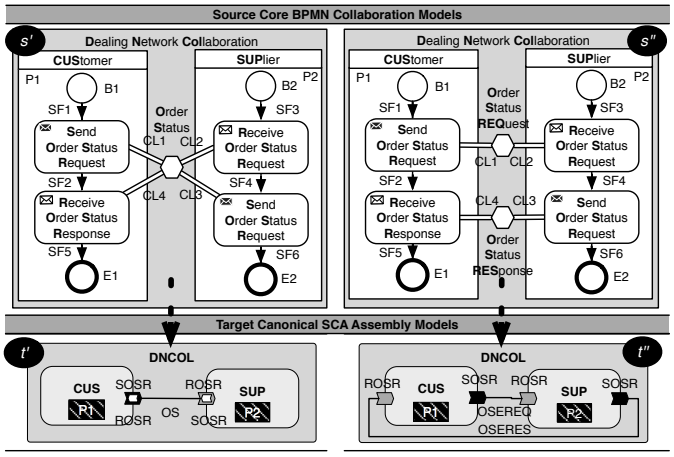

Fig. 1. Examples of BPMN collaboration and SCA assembly diagrams.

The business links between the partner processes can be modeled with a single *order status* conversation construct in s' that contains all the service interactions, or as two different conversations in s'' that logically separates the exchanged between the participants. In the collaboration model s', the business service is designed as two send/receive interaction pattern [15] that belong to a compound conversation. Whereas, in s'' the business service is designed with the *order status request* and *order status response* conversations that separate the service

interaction patterns. In both cases, the BPMN participants are transformed into the SCA components. To interact with each other the components expose provided ports, i.e., called services, and consume provided services by means of references, i.e., required ports. They are connected to each other with the help of the wires (i.e., connectors). Since, the BPMN participants play both consumer and provider roles in the collaboration s' that it contains a single conversation, then two callbacks are defined for the services and references for the generated SCA components in t'. They are configured to interact with the *os* wire. However, the produced target SCA assembly model t'' contains two different wires representing the separation of the service interaction logic. Finally, each component contains the implementation of the business process realizing the participant orchestrations. This allows appropriate technologies to be used for each component. It has to be noticed that we do not make any recommendation on the use of any implementation technology.

For the reminder, consider that the model s' has evolved to s'', and that a transformation has previously generated the model t' from s'. Then, what we require from the model synchronization is that it produces the target model t'' in Figure 1. In particular, it has to preserve the structural correctness of the used metamodels and consistently translate the alteration of the BPMN collaboration model into an update on the wiring topology of the SCA assembly model. Those models are respectively conform to the constructs of the **BPMN collaboration metamodel** and **SCA assembly metamodel** presented in Figure 2. In this figure, the simplified UML meta-class diagrams define the abstract syntax of the graphical language subsets in which the models are expressed. They represent the types of the assorted model objects, and the **models** are conform to a **metamodel**.

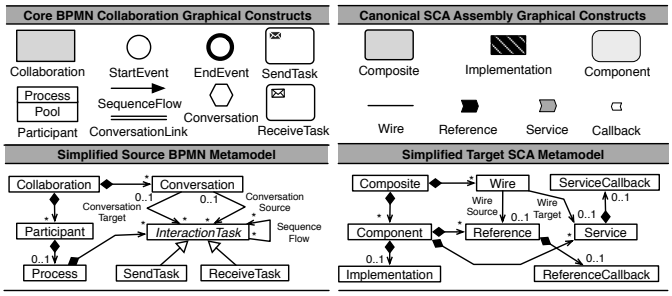

Fig. 2. BPMN and SCA assembly graphical constructs and metamodels.

B. Requirements of the Model Synchronizer

Before specifying the synchronization and its details, we need to formalize the problem of the model synchronization. Following Antkiewicz [9], let $\mathcal{S}$ and $\mathcal{T}$ be two metamodels at different levels of abstraction (e.g., the BPMN and the SCA) and $\mathcal{C} \subseteq (\mathcal{S} \times \mathcal{T})$ be the **consistency relation** established between them. This relation captures the BPMN-to-SCA conceptual mapping that we introduce in the next section. Let a source model s, i.e., which conforms to $\mathcal{S}$, and a target model t, i.e., which conforms to $\mathcal{T}$, be consistent (i.e., synchronized) with respect to the relation $\mathcal{C}$, i.e., $(s,t) \in \mathcal{C}$. Given a source update $U_{\Delta\mathcal{S}}$ that alters s to s', **the problem is to translate the update of the source $U_{\Delta\mathcal{S}}$ into a consistent update of the target $U_{\Delta\mathcal{T}}$**, such that the application of both updates results in consistent models, i.e, $(U_{\Delta\mathcal{S}}(s), U_{\Delta\mathcal{T}}(t)) \in \mathcal{C}$. The synchronizer that provides this function is defined as follows:

*Definition 1 (***Synchronizer):** A unidirectional incremental source-to-target synchronizer using update translation is a partial function of type $\mathcal{S} \times \Delta\mathcal{S} \times \mathcal{T} \rightarrow \mathcal{T}$. For $s \in \mathcal{S}, U_{\Delta\mathcal{S}}, t \in \mathcal{T}$ and $(s,t) \in \mathcal{C}$ it determines $U_{\Delta\mathcal{T}}$ such that $(U_{\Delta\mathcal{S}}(s), U_{\Delta\mathcal{T}}(t)) \in \mathcal{C}$ and then computes $t' = U_{\Delta\mathcal{T}}(t)$.

The $\Delta\mathcal{S}$ (resp., $\Delta\mathcal{T}$) is an abbreviation for the update types on the models that conform to $\mathcal{S}$ (resp., $\mathcal{T}$). They represent the space of all partial functions $\mathcal{S} \nrightarrow \mathcal{S}$ (resp., $\mathcal{T} \nrightarrow \mathcal{T}$). To make the model transformation effectively incremental, we use the so-called **update translation** operator [9]. This operator takes as a parameters the previously transformed models and the update of the source model $U_{\Delta\mathcal{S}}$. It enforces the translation $U_{\Delta\mathcal{S}}$ into $U_{\Delta\mathcal{T}}$ not through a model translation (i.e., a *diff*-based method [16]), but by synchronizing in-place (i.e., computing $U_{\Delta\mathcal{T}}(t)$) the existing target model t with the updated s'. The reason is that in the case where $U_{\Delta\mathcal{S}}$ is small, generally, it corresponds to a small target update $U_{\Delta\mathcal{T}}$, and the performance savings for the computation of t' are expected to be high [9]. Taking into account a small fractions of source and target models, the effort to determine $U_{\Delta\mathcal{T}}$ and compute $t' = U_{\Delta\mathcal{T}}(t)$ should be much less than to compute a **model translation** from s' (i.e., as a function of type $\mathcal{S} \rightarrow \mathcal{T}$). The speedup of the incremental model transformation results in a reasonable decoupling from the source model size [11]. A discussion of the properties of such a synchronizer can be found in [17].

C. The BPMN-to-SCA Conceptual Mapping

At a business level with a Service Dominant Logic, the processes are focused on the services composition in a business driven-fashion (i.e., the orchestration process), and in that sense the *business services become process activities*. Moreover, the processes become composite services that orchestrates other business services. With the BPMN, the business services can be modeled by participant constructs with explicit process definitions, and their tasks specify the business service interactions. For a modeling purpose, we consider that the (core) subset of the BPMN constructs shown in Figure 2 is sufficient to capture the service orchestration concepts. To design the service networks as BPMN collaborations, we make the following definition:

*Definition 2 (***Source Core BPMN Collaboration Model):** A source core BPMN collaboration model, i.e., assimilated to a single **collaboration**, is a tuple $(\mathcal{P}, \mathcal{A}^{\triangleright}, \mathcal{A}^{\triangleleft}, \mathcal{F}, \mathcal{M}, \mathcal{U})$ where $\mathcal{P}$ is a set of **participants** which can be partitioned into disjoint sets of **send tasks** $\mathcal{A}^{\triangleright}$, **receive tasks** $\mathcal{A}^{\triangleleft}$, $\mathcal{A} = (\mathcal{A}^{\triangleright} \cup \mathcal{A}^{\triangleleft})$ is the set of **interaction tasks**, $\mathcal{F} \subseteq \cup_{p \in \mathcal{P}}(\mathcal{A}_p \times \mathcal{A}_p)$ is a **sequence flow** relation between the tasks, $\mathcal{M} \subseteq \cup_{p,p' \in \mathcal{P}}(\mathcal{A}_p^{\triangleright} \times \mathcal{A}_{p'}^{\triangleleft})$ is a **conversation** relation, and $\mathcal{U} \subseteq (\mathcal{P} \times \mathcal{A})$ is a **process** relation.

In this definition, we intentionally hide the BPMN message flows between the interaction tasks since they can be subsumed by the conversation relation. We also consider the minimal well-formedness requirements on the process structure, e.g., there is a unique start event, there is one or more end events, and every interaction task is on a path from the start event to an end event. When mapped to the SCA, the *business services become components* with published remote interfaces. A SCA assembly is defined below.

*Definition 3 (***Canonical SCA Assembly Model):** A canonical SCA assembly model, i.e., assimilated to a single **composite**, is a tuple $(\mathcal{N}, \mathcal{O}^{\triangleleft}, \mathcal{O}^{\triangleright}, \mathcal{Q}^{\triangleleft}, \mathcal{Q}^{\triangleright}, \mathcal{W}, \mathcal{K})$ where $\mathcal{N}$ is a set of components which can partitioned into disjoint sets of **services** $\mathcal{O}^{\triangleleft}$, **references** $\mathcal{O}^{\triangleright}$, **service callbacks** $\mathcal{Q}^{\triangleleft}$, **reference callback** $\mathcal{Q}^{\triangleright}$, **implementations** $\mathcal{K}$, and $\mathcal{W} \subseteq (\mathcal{O}^{\triangleright} \times \mathcal{O}^{\triangleleft})$ is a **wire** relation.

We refer to [7] for detailed structural requirements on a SCA model that can be expressed in term of interface equivalence and inclusion [18]. Also, the readers can find there, a classification of the conceptual mapping rules between the core BPMN collaboration and the SCA meta-classes of Figure 2 and their associations. Table I presents a non-exhaustive list of those rules. Summarily, it defines an explicit mapping between the BPMN types and the SCA types. The SCA composite modularize and compose the business functions (i.e., contained in components) in a manner that is decoupled from their implementations. Consequently, the conversations between the BPMN participants can map to the SCA wires. Likewise, the processes map to the component implementations. We mainly focus on the control and message flow, and put the data flows out of this article scope. In the following, we introduce the correctness checking of the model updates propagated by the analysts. In the SCA, it is necessary to differentiate the constructs that are used to expose services, from those used for the service consumption. Refer to Section III-B for the detailed explanations on this mapping.

TABLE I
CONCEPTUAL MAPPING RULES BETWEEN THE BPMN AND THE SCA.

BPMN Meta-class	SCA Meta-classes
$Collaboration$	$Composite$
$Participant$	$Component$
$SendTask$	$Reference$ or $ServiceCallback$
$ReceiveTask$	$Service$ or $ReferenceCallback$
$Process$	$Implementation$
$Conversation$	$Wire$
$ConversationSource$	$WireSource$
$ConversationTarget$	$WireTarget$

III. INCREMENTAL MODEL TRANSFORMATION

In the previous section, we have shown that the considered models are assorted collections of typed objects and links among them. In this section, we introduce our incremental model transformation framework that uses graph rewritings.

In Definitions 2, the relation $\mathcal{F}$ defines a directed graph over the process task set $\mathcal{A}$, and the relation $\mathcal{M}$ defines a directed graph over the tasks of the interacting participant. Analogously, a SCA assembly can be seen as a directed graph of components linked through the relation $\mathcal{W}$ given in Definitions 3. Thus, a core BPMN collaboration model and a canonical SCA assembly model can be uniformly represented by directed graphs. We have made this choice for various reasons. The *Labeled nested typed rooted graphs* [10] provide an intuitive, mathematically well-understood and general formalism with a tool support to represent the model structures. The *edges* in a graph are used to represent all kinds of relations between the model objects that are represented by the *nodes* given in Definition 4. Each node and each edge has a *label* and a *type* that unambiguously identify the objects in each model. Nesting is used as a mean to reduce the graph complexity, by allowing nodes to contain other nodes.

*Definition 4 (***Model Assimilated to a Graph):** Let a model be a set of objects and links that can be partitioned into disjoint sets of node identifiers $\mathcal{V}$ and edge identifiers $\mathcal{E}$. It is assimilated to a tuple $(\mathcal{V}, \mathcal{E}, \mathcal{B}, \mathcal{Y}, \mathcal{I}, \perp, label, type, source, target, parent, child)$ where $\mathcal{B}$ is the set of node labels, $\mathcal{Y}$ is the set of nodes and edge types, $\mathcal{I}$ a set of indexes for the ordered object nesting, $\perp$ is a root node identifier (i.e., the model container), and the functions $label : \mathcal{V} \rightarrow \mathcal{B}$, $type : \mathcal{V} \cup \mathcal{E} \rightarrow \mathcal{Y}$, $source, target : \mathcal{E} \rightarrow \mathcal{V}$, $parent : \mathcal{V} \rightarrow \mathcal{V}$, $child : \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{I}$ define the graph structure. $(label(\perp) = type(\perp) = parent(\perp) = \perp)$

The Figure 3 shows an example of two graph structure for a core BPMN collaboration model, e.g., s, and a SCA assembly model, e.g., t. The node and the links in the graphs are labeled with the label's first letters of the models. The model s is represented by the graph G_s and the model t by the graph G_t. The containers of the graph representing the BPMN model is denoted $\perp_s$, and $\perp_t$ denotes the container of the graph G_t.

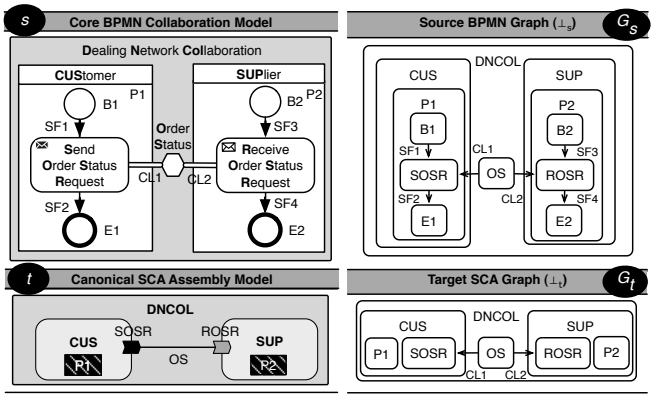

Fig. 3. Graphs for BPMN collaboration and SCA assembly models.

Through the case study and the previous definitions, we have introduced a graph structure to describe the source and the target models. In the next section, we explain how the updates on the considered models are expressed with the help of the conditional graph rewriting.

A. Conditional Graph Rewriting

The *conditional graph rewriting* [11], [3] approaches has been intensively explored for expressing software artifacts updates. In this paper, we are particularly interested in treating complex sequences of graph production as model updates. However, in order to guarantee the correctness of a model update there is no alternative than to reduce its granularity into primitive updates. Otherwise, the problem becomes unsolvable, as it is inherently compounded with the combinatorial changes of the model's object patterns. An update describes the change of model's internal structure. We consider that the application of a structural model update correspond to the execution of a sequence of primitive graph productions such as node and edge additions, removals, relocations operations [12]. For example, Table II shows a list of such primitive operations. One way for obtaining a sequence of primitive update operations is by recording editing operations performed by the business analysts on the BPMN models.

TABLE II
PRIMITIVE GRAPH PRODUCTIONS.

Primitive Change Operation	Description
$addNode(v, b, y, v', i)$	Adds the node v of a type y in the parent node v' with a label b and an index i
$insertEdge(e, v, v', b, y)$	Inserts the edge e of a type y with a label b between the nodes v and v'
$dropNode(v, v')$	Drops an existing node v from the parent node v'
$deleteEdge(e, v, v')$	Deletes an existing edge e between the nodes v and v'
$setLabel(v, b, b')$	Sets the label of the node v from b to b'
$setSource(e, v, v')$	Sets the source of the edge e from v to v'
$setTarget(e, v, v')$	Sets the target of the edge e from v to v'
$setIndex(v, i, i', v')$	Sets the index of the node v from i to i' in the parent node v'

For instance, an addition of a process fragment into a BPMN collaboration model can be seen as a composite production of several graph productions (i.e., primitive operations). For example, the Figure 4 depicts the operations that produce the models of Figure 3. The graph productions $U_{\Delta\mathcal{S}}$ construct the graph G_s representing the source core BPMN collaboration model s. The graph productions $U_{\Delta\mathcal{T}}$ construct the graph G_t representing the target core SCA assembly model t. Consider the operation $addNode(5, SOSR, SendTask, 3, init)$ in $U_{\Delta\mathcal{S}}$. It adds the send task node to the process $P1$, and for example, $insertEdge(17, CL1, ConversationSource, 16, 5)$ inserts an edge between the conversation OS and the former $SOSR$ send task. Note that we assume the well-behavedeness of the updates propagated by the business analysts. It means that the graph production on a core BPMN collaboration model are consistent with the behavioral requirements as defined in [15].

B. Consistency Management

In Section II-B, we stated that a *consistency relation*, denoted $\mathcal{C}$, has to be established between the subsets of BPMN and SCA metamodels (i.e., the model object types). It is guaranteed by a first transformation and should be preserved through the change propagation. In practice, the relation $\mathcal{C}$ partially covers $\mathcal{S}$ or $\mathcal{T}$ because the BPMN and the SCA are conceptually different. This means that, when the transformation take place, only the BPMN objects with types that has proper conceptual mapping into the SCA types are transformed. For example, in Figures 1 and 3 only the BPMN constructs that specify service interactions are transformed into elements of the core SCA assembly model. The other process constructs such as events and sequence flows (i.e., see Figure 2) are not relevant in the generation of the canonical SCA topology [7]. For this purpose, we introduce an explicit structural correctness relation between the node and edge types of the assorted models. This relation should be used for the assessment of the graph production application.

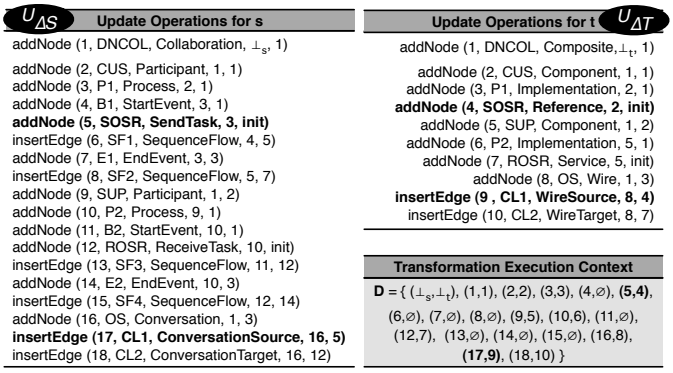

Fig. 4. Primitive graph productions for the models of Figure 3.

*Definition 5 (***Structural Correctness):** The structural correctness for a model assimilated to a graph, i.e., given in Definition 4, is a binary relation $\mathcal{Z} = \{(y, y') \in \mathcal{Y} \times (\mathcal{Y} \cup \perp) \mid \exists v, v' \in (\mathcal{V} \cup \perp), i \in \mathcal{I}, type(v) = y \wedge type(v') = y' \wedge cons(y) = y' \wedge \underline{child(v') = (v, i)}\} \cup \{(y, y') \in \mathcal{Y} \times \mathcal{Y} \mid \exists e \in \mathcal{E}, v, v' \in \mathcal{V}, i \in \mathcal{I},\ type(e) = y \wedge type(v) = y' \wedge type(v') = y'' \wedge cons(y) = y' \wedge cons(y') = y'' \wedge \underline{(source(e) = v \vee target(e) = v) \wedge child(v') = (v, i)}\} \cup \underline{\{(\perp, \emptyset)\}}$ where $cons : \mathcal{Y} \rightarrow (\mathcal{Y} \cup \perp)$ is the constraint function between the model types.

For example, Table III shows the structural correctness relations for a core BPMN collaboration s and a canonical SCA assembly models t as tuples of the model types defined in Definitions 2 and 3. The relations $\mathcal{Z}_s$ and $\mathcal{Z}_t$ express the imbrications between the types depicted in Figure 2, or the associations between them, i.e., their names are abbreviated in the table. For instance, $(Part., Collab.)$ in $\mathcal{Z}_s$ means that all node typed with $Participant$ must be contained in nodes typed with $Collaboration$. Also, $(W.Tar, Serv.)$ and $(W.Sour., W.)$ in $\mathcal{Z}_t$ mean that edges typed with $WireTarget$ should have nodes with $Service$ type as a source and a $Wire$ as target.

Furthermore, to assess a consistent incremental model transformation each node or edge type must hold in the consistency relation $\mathcal{C}$. This relation is given in Definition 6. It is the invariant of the transformation between the source and target models. The simplified consistency relation $\mathcal{C}$ shown in Table III transcribes the BPMN-to-SCA conceptual mapping rules presented in Table I. For example, the tuple $((Conv., Collab., \emptyset), (Wire, Composite))$ mean that source nodes typed with $Conversation$ and contained in a collaboration should be

TABLE III
SIMPLIFIED STRUCTURAL CORRECTNESS AND CONSISTENCY RELATIONS.

Core BPMN collaboration structural correctness: $\mathcal{Z}_s$
$(Collab., \perp)$, $(Part., Collab.)$, $(Proc., Part.)$, $(SendTask, Proc.)$, $(Rec.Task, Proc.)$, $(Seq.F.Sour., Int.Task)$, $(Seq.F.Tar., Int.Task)$, $(Conv., Collab.)$, $(Conv.Tar., Conv.)$, $(Conv.Sour., Conv.)$, $(Conv.Sour., Int.Task)$, $(Conv.Tar., Int.Task)$, $(\perp, \emptyset)$
Canonical SCA assembly structural correctness: $\mathcal{Z}_t$
$(Compos., \perp)$, $(Compon., Compos.)$, $(Impl., Compon.)$, $(Ref., Compon.)$, $(Serv., Compon.)$, $(Serv.Callb., Service)$, $(Ref.Callb., Ref.)$, $(W., Compos.)$, $(W.Tar., W.)$, $(W.Sour., W.)$, $(W.Tar., Serv.)$, $(W.Sour., Ref.)$, $(\perp, \emptyset)$
BPMN-to-SCA consistency relation: $\mathcal{C}$
$((Collab., \perp, \emptyset), (Compos., \perp))$, $((Part., Collab., \emptyset), (Compon., Compos.))$, $((Proc., Part., \emptyset), (Impl., Compon.))$, $((Conv., Collab., \emptyset), (W., Compos.))$, $((SendTask, Proc., init), (Ref., Component))$, $((Rec.Task, Proc., init), (Ser., Component))$, $((Rec.Task, Proc., fin), (Ref.Callb., Ref.))$, $((SendTask, Proc., fin), (Serv.Callb., Serv.))$, $((Conv.Sour., Conv., \emptyset), (W.Sour., W.))$, $((Conv.Tar., Conv., \emptyset), (W.Tar., W.))$, $((Conv.Sour., SendTask, init), (W.Sour., Ref.))$, $((Conv.Tar., Rec.Task, init), (W.Tar., Serv.))$, $((\perp, \emptyset, \emptyset), (\perp, \emptyset))$

mapped to wires that are contained in a composite. Actually, the types in the respective BPMN or SCA models of the previous figures hold in the correctness relations: $\mathcal{Z}_s$ and $\mathcal{Z}_t$.

*Definition 6 (***Consistency Relation):** The consistency relation established between two models $s = (..., \mathcal{Y}_s, \mathcal{I}_s, ...)$ and $t = (..., \mathcal{Y}_t, \mathcal{I}_t, ...)$ (i.e., assimilated to a graphs) is a relation $\mathcal{C} = \{\underline{((y, y', i), (y'', y'''))} \in (\mathcal{Y}_s \times \mathcal{Y}_s \times \mathcal{I}_s) \times (\mathcal{Y}_t \times \mathcal{Y}_t) \mid (y, y') \in \mathcal{Z}_s \wedge (y'', y''') \in \mathcal{Z}_t \wedge map(y, i) = (y'', i) \wedge map(y', i') = (y''', i')\}$ where $\underline{map : (\mathcal{Y}_s \times \mathcal{I}_s) \rightarrow (\mathcal{Y}_t \times \mathcal{I}_t)}$ is the partial function that conceptually maps the model types.

Note that not all the BPMN tasks are straightly mapped to the SCA ports. In the SCA, the component ports are differentiated into services or references [14]. Each of them characterizes the direction of the first exchanged message between two components. The invoker of the first service call through a reference is considered to be the consumer of a service of another (provider) component. Thus, in order to correctly compose the SCA components that play both provider and consumer roles (i.e., with callbacks), it is necessary to determine which BPMN task sends the first and the last messages. A simple static analysis of the process flow is considered to distinguish tasks by outgoing or incoming message interactions with the other participants. For this purpose, we define an *index* function as $\forall p, p' \in \mathcal{P}, \exists a \in \mathcal{A}_p, a' \in \mathcal{A}_{p'}$, $\underline{(a, a') \in \mathcal{M}}$ we have $\underline{index(a) = init}$, if $\nexists a'' \in \mathcal{A}_p \mid a'' \in prev^*(a)$, and $\underline{index(a) = fin}$, if $\nexists a'' \in \mathcal{A}_p \mid a'' \in next^*(a)$. The functions denoted $prev^*(a) = \{a' \in \mathcal{A}_p \mid (a, a') \in \mathcal{F}^*\}$ and $next^*(a) = \{a' \in \mathcal{A}_p \mid (a, a') \in \mathcal{F}^*\}$ give the set of all direct and transitive predecessors and successors of the task a, where $\mathcal{F}^*$ is the reflexive transitive closure of $\mathcal{F}$.

By introspecting the indexes of the interaction tasks, it is possible to find the tasks which capture the service interaction points. A first receive task, i.e., with an index "$init$", is mapped to a service. This is expressed by the tuple $((SendTask, Proc., init), (Ref., Component))$ in $\mathcal{C}$. Accordingly, a first send task maps to a reference. Obviously, the processes can contain a range of other send tasks (resp., receive tasks) after the first receive task (resp., a first service task) to interact with the same participant. In this case, a send task (resp., a receive task) with an index "fin" is mapped to a callback and it is associated with the first mapped reference (resp., the first service), i.e., $((SendTask, Proc., fin), (Serv.Callb., Serv.))$. The other tuples such $((Conv.Sour., SendTask, init), (W.Sour., Ref.))$ and $((Conv.Tar., Conv., \emptyset), (W.Tar., W.))$ capture the mapping between edges that are attached to $Conversation$ and $Wire$.

Since the relation $\mathcal{C}$ is not functional, then, there maybe multiple canonical SCA transformations of the same BPMN model, and different BPMN models can be implemented by the same SCA model. Also, the transformation may results in empty target models due to the lack of the source model expressiveness. In order to preserve the **transformation execution context**, we introduce a binary relation between the identifier of the related models as $\mathcal{D} = \underline{\{(\perp_s, \perp_t)\}} \cup \{\underline{(v, v') \in \mathcal{V}_s \times \mathcal{V}_t} \mid \exists v'' \in \mathcal{V}_s, i \in \mathcal{I}_s, v''' \in \mathcal{V}_t \mid label_s(v) = label_t(v') \wedge child_s(v) = (v'', i) \wedge child_t(v') = (v''', i) \wedge ((type_s(v), type_s(v''), i)), (type_t(v'), type_t(v''')))) \in \mathcal{C}\} \cup \{\underline{(e, e') \in E_s \times E_t} \mid \exists v, v' \in \mathcal{V}_s, v'', v''' \in \mathcal{V}_t, i, i' \in \mathcal{I}_s \mid label_s(e) = label_t(e') \wedge source_s(e) = v \wedge target_s(e) = v' \wedge source_t(e') = v'' \wedge target_t(e') = v'''' \wedge ((type_s(e), type_s(v), i), (type_t(e'), type_t(v''))), ((type_s(e), type_s(v'), i'), (type_t(e'), type_t(v'''))) \in \mathcal{C}\}$. It is used by our trace-aware synchronizer for the incremental transformation. For example, consider the transformation execution context of Figure 4. This relation represents the structural correspondence between the BPMN model and its related SCA model of Figure 3.

C. Model Synchronization Algorithm

According to the requirement of Definition 1, let a target canonical SCA assembly model t being previously obtained from a transformation of source core BPMN collaboration model s. Given a well-formed production sequence on the source model, i.e., $U_{\Delta\mathcal{S}}$, that alters s to s', first, our synchronizer locates the corresponding constructs in the transformation execution context $\mathcal{D}$ that are affected by each change operation δ in $U_{\Delta\mathcal{S}}$. The synchronizer, shown in Algorithm 1, works by translating the primitive update operations on s into updates of the target SCA model t. Then, with the identified constructs correspondence it translates the operation δ (i.e., given in Table II) into a well-formed update $U_{\Delta\mathcal{T}}$ on the target w.r.t consistency relation $\mathcal{C}$. Finally, rather than "executing the entire transformation afresh", the synchronizer computes in-place a new model t' (i.e., $(U_{\Delta\mathcal{S}}(s), t') \in \mathcal{C}$) as an alteration of t by applying the obtained $U_{\Delta\mathcal{T}}$.

To illustrate the synchronization algorithm, consider two models $\{\perp_s\}$ and $\{\perp_t\}$ that are synchronized, and a transformation context $\mathcal{D} = \{(\perp_s, \perp_t)\}$. Then, given the update operation sequence $U_{\Delta\mathcal{S}}$ of Figure 4 that produces the BPMN model s of Figure 3, the operator translates it into the well-formed update operation $U_{\Delta\mathcal{T}}$ sequence for t in the same figure. For instance, the operation $addNode(5, SOSR, SendTask, 3, init)$ on s is translated to the operation $addNode(4, SOSR, Reference, 2, init)$ on t, and $\mathcal{D}$ is updated with the couple $(5, 4)$ in the line *1* of the Algorithm 1. However, when an operation adds a node with

Algorithm 1: Synchronization Algorithm.

input : $s = (..., \mathcal{I}_s, ..., type_s, ..., target_s, ...)$, $U_{\Delta\mathcal{S}}$, $t = (\mathcal{V}_t, \mathcal{E}_t, ..., \mathcal{Y}_t, ..., type_t, source_t, target_t, parent_t, ...)$ /* **With** $(s,t) \in \mathcal{C}$ */
output: t'
begin
  $U_{\Delta\mathcal{T}} \leftarrow \emptyset$; /* $U_{\Delta\mathcal{S}}$ **is well-formed** */
  foreach $\delta \in U_{\Delta\mathcal{S}}$ **do**
1 **if** $\delta =$ ***addNode(v, b, y, v', i)*** $\wedge\ \exists v''' \in \mathcal{V}_t \mid (v', v''') \in \mathcal{D} \wedge$ $y' \in \mathcal{Y}_t \mid ((y, type_s(v'), i), (y', type_t(v''')))) \in \mathcal{C}$ **then**
      $\exists v'' \in \mathcal{V}_t \mid \mathcal{D} \leftarrow \mathcal{D} \cup (v, v'')$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***addNode(v'', b, y', v''', i)***;
    else $\mathcal{D} \leftarrow \mathcal{D} \cup (v, \emptyset)$;
    if $\delta =$ ***insertEdge(e, v, v', b, y)*** $\wedge$ $\exists v'', v''' \in \mathcal{V}_t \mid (v, v''), (v', v''') \in \mathcal{D} \wedge$ $\exists y' \in \mathcal{Y}_t, i, i' \in \mathcal{I}_s \mid ((y, type_s(v), i), (y', type_t(v''))),$ $((y, type_s(v'), i'), (y', type_t(v''')))) \in \mathcal{C}$ **then**
      $\exists e' \in \mathcal{E}_t \mid \mathcal{D} \leftarrow \mathcal{D} \cup (e, e')$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***insertEdge(e', v'', v''', b, y')***;
    else $\mathcal{D} \leftarrow \mathcal{D} \cup (e, \emptyset)$;
    if $\delta =$ ***dropNode(v, v')*** $\wedge\ \exists v'', v''' \mathcal{V}_t \mid (v, v''), (v', v''') \in \mathcal{D} \wedge$ $\exists i \in \mathcal{I}_s \mid$ $((type_s(v), type_s(v'), i), (type_t(v''), type_t(v''')))) \in \mathcal{C}$ **then**
      $\mathcal{D} \leftarrow \mathcal{D} \setminus (v, v')$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***dropNode(v'', v''')***;
    if $\delta =$ ***deleteEdge(e, v, v')*** $\wedge$ $\exists e' \in \mathcal{E}_t, v'', v''' \in \mathcal{V}_t \mid (e, e')(v, v''), (v', v''') \in \mathcal{D} \wedge$ $\exists i, i' \in \mathcal{I}_s \mid ((type_s(e), type_s(v), i), (type_t(e'), type_t(v''))),$ $((type_s(e), type_s(v'), i'), (type_t(e'), type_t(v''')))) \in \mathcal{C}$ **then**
      $\mathcal{D} \leftarrow \mathcal{D} \setminus (e, e')$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***deleteEdge(e', v'', v''')***;
    if $\delta =$ ***setLabel(v, b, b')*** $\wedge\ \exists v' \in \mathcal{V}_t \mid (v, v') \in \mathcal{D}$ **then**
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***setLabel(v', b, b')***;
    if $\delta =$ ***setSource(e, v, v')*** $\wedge$ $\exists v'', v''' \in \mathcal{V}_t \mid (e, \emptyset), (v', v''), (target_s(e), v''') \in \mathcal{D} \wedge$ $\exists y \in \mathcal{Y}_t, i, i' \in \mathcal{I}_s \mid$ $((type_s(e), type_s(v'), i), (y, type_t(v''))),$ $((type_s(e), type_s(target_s(e)), i'), (y, type_t(v''')))) \in \mathcal{C}$ **then**
      $\exists e' \in \mathcal{E}_t \mid \mathcal{D} \leftarrow \mathcal{D} \cup (e, e')$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***insertEdge(e', v'', v''', label$_s$, y)***;
    else if $\exists e' \in \mathcal{E}_t, v'' \in \mathcal{V}_t \mid (e, e'), (v', v'') \in \mathcal{D} \wedge \exists i \in \mathcal{I}_s \mid$ $((type_s(e), type_s(v'), i), (type_t(e'), type_t(v''))) \in \mathcal{C}$ **then**
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***setSource(e', source$_t$(e'), v'')***;
    else
      $\mathcal{D} \leftarrow \mathcal{D} \setminus (e, e') \cup (e, \emptyset)$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***deleteEdge(e', source$_t$(e'), target$_t$(e'))***;
    /* **Similar instructions for the** $setTarget(e, v, v')$ **operation** */
2 **if** $\delta =$ ***setIndex(v, i, i', v')*** $\wedge$ $\exists v'', v''' \in \mathcal{V}_t \mid v''' \neq parent_t(v'') \wedge$ $(v, v''), (v', v''') \in \mathcal{D} \wedge \exists y \in \mathcal{Y}_t \mid y \neq type_t(v'') \wedge$ $((type_s(v), type_s(v'), i'), (y, type_t(v'''))) \in \mathcal{C}$ **then**
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup \{$ ***dropNode(v'', parent$_t$(v''))***, ***addNode(v'', label$_t$(v''), y, v''', i')***$\}$;
    else
      $\mathcal{D} \leftarrow \mathcal{D} \setminus (v, v'') \cup (v, \emptyset)$;
      $U_{\Delta\mathcal{T}} \leftarrow U_{\Delta\mathcal{T}} \cup$ ***dropNode(v'', parent$_t$(v''))***;
  $t' \leftarrow U_{\Delta\mathcal{T}}(t)$; /* **With** $(U_{\Delta\mathcal{S}}(s), t') \in \mathcal{C}$ */

a type that does not hold in the consistency relation such as $addNode(7, E1, EndEvent, 3, 3)$, $\mathcal{D}$ is updated with $(7, \emptyset)$. It means that this object has no counterpart SCA construct. Second, consider the update sequence $U'_{\Delta\mathcal{S}}$ of Figure 5 for s' of Figure 1 in which the addition of the receive task $ROSR$ is similarly translated onto an addition of a callback of the perviously generated reference $SOSR$ in $U'_{\Delta\mathcal{T}}$. The transformation context translation is updated with the couple $(19, 11)$ of those object identifiers. Also, updating of the index of $ROSR$ receive task in $U''_{\Delta\mathcal{S}}$ with the operation is translated into a deletion (i.e., line *2* of the Algorithm 1) of the corresponding reference callback and the addition of the service $ROSR$ to the component SUP in the sequence $U''_{\Delta\mathcal{T}}$. Finally, without disrupting the core SCA assembly model consistency and completely re-generating the model t'' of Figure 1, the system architects can rapidly assess the necessity to propagate the changes to the SCA implementations.

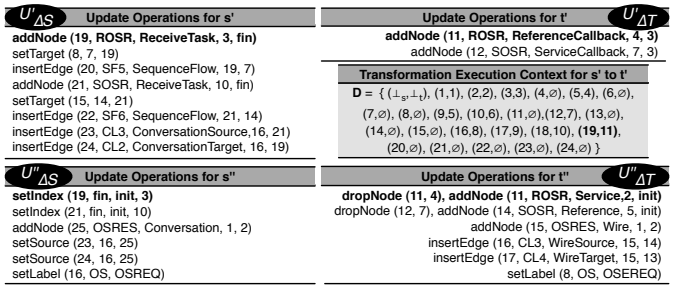

Fig. 5. Composite graph productions for the models of Figure 1.

This section described our update translation operator for managing and re-establishing the structural consistency. Moreover, adopting a MDE approach to build the service-enabled software solutions may require the usage of different technologies to implement the SOA. Also, the composite applications can contain both business services newly created for the application, and business function from existing system applications, reused as part of the composition. Due to lack of space, we intentionally put the specification of those structural constraints (e.g., the prevention of relevant component update, or the choice on the implantation technology) out of the scope of this paper.

D. Model Synchronization Implementation

For a proof-of-concept prototype, the synchronizer has been successfully implemented as an extension to the *Yaoqiang BPMN Editor* [19]. This editor is compliant with BPMN 2.0 specification [13] and it is based on the *JGraph* [20] graph visualization library. We added a filter that logs the primitive graph editing operation which manipulate the BPMN construct types shown in Figure 2. We have also implemented a removing redundancy algorithm that transforms a sequence of graph structure operations into a normalized form as specified in [12]. With the synchronization algorithm, we obtain a graph editing operation sequence that manipulates SCA constructs types. This sequence again enforced with the *JGraph* library is applied to another graph structure representing the SCA assembly model. The development of an editor to visualize the resulting SCA model is a work in progress.

A fundamental transformation law is the determinism. In our case, the transformation is modeled as a mathematical function. It means that given the same pair of models, the update translation algorithm always returns the same proposed modification. The synchronizer given in Algorithm 1 propagates only the operations that synchronize the SCA model with the BPMN model. With these properties, we can conclude

that the synchronizer is terminating and locally confluent. We refer the reader to [12], [11] for further discussions on the scalability measurements. The current implementation checks for the applicability conflicts of our primitive change operations. Evidently, that there are some limitations inherent to any operation-based approach in comparison with a state-based one. The introduced operations can be composed to give rise to model update types with enhanced semantics for the users (e.g., splitting a service, assigning a task to a component through their behavioral concerns). However, we restricted our presentation to the structural correctness. For example, the semantic that means that a business process update is a modification in the service interaction pattern is out of the scope of this paper.

IV. Related Research

The BPM [6] used within the MDE approaches [7], [8] often results in SOA models that are automatically transformed from the business process models. However, by the time a business process evolution must be implemented in the IT infrastructure on the opportunity to capitalize on the previous SOA implementations may be lost. Further, responding to the business process evolution is still a manual practice. In the SOA space, the MDA [21] enables business level functionalities to be modeled with the UML [22]. Certainly, the SOA models can be expressed using the UML [23]. However, as the UML is neither fully service-centered, nor process-oriented [22], it is necessary to provide alternative design methods. The standard BPMN [13] offers the sufficient conceptual information needed to specify a service-enabled software solution in cross-organizational settings. In order to define the business process logic of each partner pertaining to a domain being modeled, the BPMN refers to a service orchestration [1] as a participant's private process. However, it neither specifies the roles (e.g., service consumer or provider) that participants are expected to play in a business collaboration, nor foreshadow the architecture of the supporting SOA implementation. The SCA specifications [14] provide a framework for creating software solutions in a multi-language and multi-platform environment that are organized in a SOA. Unlike the BPMN, the SCA describes the system configuration (or run-time architectures) and its deployment. It is supported by an integration middleware that provides the broker and the binding mechanisms which follow the SOA principles.

The need for synchronizing multiple heterogeneous software languages arises in multi-level SOA development. The *multi-level incremental change propagation* and the *change impact analysis* in the SOA development was studied in [24], [25]. Separately, other approaches have studied the reconfiguration on the component models [3], [26]. However, to the best of our knowledge only the *general model synchronization* frameworks [9] was used for the consistency management among those heterogenous models. Those frameworks can guarantee the consistency between software artifacts that are expressed in different languages. However, they require to manually write the synchronization code to deal with each update kind on each of the assorted models [17]. Compared to these approaches, our synchronizer automatically extracts the synchronization informations from the previous transformation context and does not require the users to write code to handle for model modifications.

The term "synchronization" sometimes refers to the approaches to differencing and merging models that conform to the same metamodel [16]. These approaches can be used in the MDE to merge models with multiple concurrent updates. Unfortunately, they lack computations with reasonable decoupling from the model sizes [11]. Moreover, the language-specific syntactic consistency of the assorted model types can be formally expressed in *OCL-compliant notation* in the transformation code [25]. However, this makes the synchronization code much complex, particularly, when not all the source metamodel maps to the target one. In this case, the generality of the incremental model synchronization system cannot fit in here. The promising formal *conditional graph rewriting* approaches [3], [11] has been intensively explored to express automatic software artifacts transformations in a uniform way. However, the change propagation across *heterogeneous software models* still remains as an open problem [17]. In [12], the *composite graph productions* were used to express the software evolutions. However, this technique is mainly focused on the homogeneous model transformation, i.e., where the source and target models conform to the same metamodel. Our work performs a more through investigation on the heterogeneous model synchronization while preserving the respective SCA and BPMN model consistency.

V. Summary and Outlook

The automation of a (business process driven) system engineering that adheres to a SOA architectural style needs to go beyond of identifying the system configurations that have to be implemented by the developers. Since this is now performed through a MDE approach, what is required from the business analysts is that they design the process activities and their dependencies with the external business services as task centric flows. In the first development stages, the BPM often results in models that are conceptually different the SOA implementations. In order to bridge this gap, we introduce a BDD method in which we start with the business requirements contained in BPMN models and work our way towards a seamless transformation into software architectures that fulfill the business goals. In summary, our approach aims at maintaining the alignment between a (source) BPMN model and a (target) SCA model which is organized in a SOA. When a target model has been generated (and implemented) and that the business requirements evolve, our synchronizer enforces automated in-place translations of correct updates on the BPMN model into correct operations on the SCA. Then, without disrupting the system structural consistency and rather than executing the entire transformation afresh, the system architects can assess the necessity to propagate the design changes to the run-time implementation. Finally, the IT developers can synchronize the application code in order

to maintain the Business/IT alignment. Furthermore, since separation occurs at the level of models rather than at the level of opaque code, this enables quick and localized evolutions of the SOA implementations in response to the business changes.

The experimental results are encouraging. A proof-of-concept prototype that supports the synchronization algorithm have been successfully implemented. The approach seems very promising, but still needs to be validated in a real-scale case study and tool. The integration of our update translation operator in a previously developed *ATLAS Transformation Language* [8] chain for the *Eclipse SOA Tools Platform* [7] is in development. Furthermore, the full round-tripping between business processes and the software architectures is essential to rapidly realign the process to accommodate changing business conditions. Consequently, a technical challenge is the bidirectional transformation of their concurrent updates. Likewise, we have to consider update to both the business and IT architectures via pattern-based techniques. Striving for further alignment, and making SOA development more tractable to reconfigure architectures without disrupting the functional capabilities of the implementations remain as a future work. Of course, the BPMN expressiveness is higher than this of the SCA, because there is no mean of control-flows between the SCA components. The BPMN behavioral aspects (e.g., tasks, gateways, flows) have no direct counterparts in the SCA. Therefore, we can enlarge our approach by transforming BPMN control-flows and message interactions to non-standard SCA annotations.

REFERENCES

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[2] H. Karhunen, M. Jantti, and A. Eerola, "Service-oriented software engineering (sose) framework," in *Services Systems and Services Management, 2005. Proceedings of ICSSSM '05. 2005 International Conference on*, vol. 2, Jun. 2005, pp. 1199 – 1204 Vol. 2.

[3] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (Re)configuration language," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-9, vol. 26, no. 5. New York, NY, USA: ACM, September 2001, pp. 21–32.

[4] 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," *IEEE Trans. Serv. Comput.*, vol. 3, pp. 145–160, April 2010.

[5] K. Levi and A. Arsanjani, "A goal-driven approach to enterprise component identification and specification," *Commun. ACM*, vol. 45, no. 10, pp. 45–52, 2002.

[6] M. Dumas and T. Kohlborn, "Service-enabled process management," in *Handbook on Business Process Management 1*, ser. International Handbooks on Information Systems, P. Bernus, J. Blazewics, G. Schmidt, M. Shaw, J. v. Brocke, and M. Rosemann, Eds. Springer Berlin Heidelberg, 2010, pp. 441–460.

[7] K. Dahman, F. Charoy, and C. Godart, "Generation of Component Based Architecture from Business Processes: Model Driven Engineering for SOA," in *The 8th IEEE European Conference on Web Services*, Ayia Napa, Greece, 2010.

[8] D. Lopes, S. Hammoudi, J. Bézivin, and F. Jouault, "Generating transformation definition from mapping specification: Application to web service platform," in *CAiSE'05*, 2005, pp. 309–325.

[9] M. Antkiewicz and K. Czarnecki, "Generative and transformational techniques in software engineering ii," in *Design Space of Heterogeneous Synchronization*, R. Lämmel, J. Visser, and J. a. Saraiva, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 3–46.

[10] E. Biermann, C. Ermel, and G. Taentzer, "Precise Semantics of EMF Model Transformations by Graph Transformation," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, ser. MoDELS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 53–67.

[11] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software and Systems Modeling*, vol. 8, no. 1, pp. 21–43, February 2009.

[12] T. Mens, "Transformational Software Evolution by Assertions," in *CSMR Workshop on Formal Foundations of Software Evolution, Lisbon*, Mar. 2001.

[13] *Business Process Model and Notation (BPMN) 2.0, Beta 1*, OMG, May 2009.

[14] *SCA Assembly Model Specification 1.1*, Open SOA, Mar. 2009.

[15] W. van der Aalst, A. Mooij, C. Stahl, and K. Wolf, "Service Interaction: Patterns, Formalization, and Analysis," in *Formal Methods for Web Services*, ser. Lecture Notes in Computer Science, M. Bernardo, L. Padovani, and G. Zavattaro, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2009, vol. 5569, ch. 2, pp. 42–88.

[16] P. Konemann, "Model-independent differences," in *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, ser. CVSM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 37–42.

[17] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 164–173.

[18] M. Léger, T. Ledoux, and T. Coupaye, "Reliable Dynamic Reconfigurations in a Reflective Component Model," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010, vol. 6092, ch. 5, pp. 74–92.

[19] "Yaoqiang bpmn editor," (accessed 24 Feb 2011). [Online]. Available: http://bpmn.yaoqiang.org/

[20] "Yaoqiang bpmn editor," (accessed 24 Feb 2011). [Online]. Available: http://www.jgraph.com/

[21] R. Gitzel, *Model-driven Software Development Using a Metamodel-based Extension Mechanism for Uml*. Peter Lang Publishing, 2006.

[22] R. Ravichandar, N. C. Narendra, K. Ponnalagu, and D. Gangopadhyay, "Morpheus: Semantics-based incremental change propagation in soa-based solutions," *Services Computing, IEEE International Conference on*, vol. 1, pp. 193–201, 2008.

[23] T. Stahl and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*. Chichester, UK: Wiley, 2006.

[24] L.-J. Zhang, A. Arsanjani, A. Allam, D. Lu, and Y.-M. Chee, "Variation-oriented analysis for soa solution design," *Services Computing, IEEE International Conference on*, vol. 0, pp. 560–568, 2007.

[25] I. Ivkovic and K. Kontogiannis, "Tracing evolution changes of software artifacts through model synchronization," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 252–261.

[26] J. Fiadeiro and A. Lopes, "A Model for Dynamic Reconfiguration in Service-Oriented Architectures," in *Software Architecture*, ser. Lecture Notes in Computer Science, M. Babar and I. Gorton, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2010, vol. 6285, ch. 8, pp. 70–85.