Abstract – Composite Web services are built as workflows made up of component Web services. WS-BPEL has become the de facto standard used for the development of above mentioned workflows. Apart from introducing mainstreamlines related to SOA architectures and WS-BPEL language, the paper identifies some crucial issues that must be solved during composite Web service development: granularity aggregated inside of component Web services, defining a framework for invocation regardless of Web service provider, version management, discovery of component Web services, communication among component Web services and legacy system integration. They are candidates for future research. Only some of the existing research activities in mentioned crucial areas are illustrated in the paper due to space limitations.

Since WS-BPEL syntax is XML based, there are lots of models for easier development of orchestrations. Some of the existing models for model driven development of orchestrations are introduced in the latter part of the paper.

Keywords: composite Web service development; orchestration; model driven development

I. INTRODUCTION

SOA can be defined as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. Distributed capabilities in SOA systems are delivered by the mechanisms called services. A service is a function that is well-defined, self-contained, and does not depend on the context or state of other services [1].

Web services are the most frequent implementation of services. They are accepted in such degree so a concept of services often refers automatically to Web services without any explicit emphasis. W3C defines Web services in the following way. “A Web service is a software system designed to support interoperable, machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description, using SOAP messages, typically conveyed using HTTP with an XML serialization, in conjunction with other Web-related standards.”

A concept of architecture based on services has brought many advantages. It has brought higher level of segmentation. Now fragments of functionality are wrapped inside of standalone, interoperable and stateless entities and used when needed. Not only fine-grained code pieces can be re-used, but coarse-grained units are prone to be reused in diverse communication scenarios. Reuse represents a main contribution of SOA infrastructure. Loose-coupling nature is the second advantage of SOA infrastructure. A loosely coupled service-oriented system is easy and cheap to evolve and it has the potential to grow as rapidly and scale as easily as the Web [2]. Interoperability is a third advantage of SOA systems. It has been delivered with wide adoption of XML-based protocols. All of them will be described later in the paper. A fourth contribution of SOA paradigm is an easy integration of diverse applications into processes. Integration mechanisms implement intra-corporative (Enterprise Application Integration, EAI) as well as inter-corporative integration (Business to Business, B2B).

The paper is organized as follows. Section II introduces a high level perspective of reference architecture for SOA systems [3]. Since standardization is tightly related to SOA, section III introduces a possible SOA protocol stack. Section IV inspects main constructs of WS-BPEL language since it has become the de-facto standard used for development of orchestrations. Critical research areas related to orchestrations are identified in section V. Those areas are candidates for future research contributions. Section VI defines models for easier development of WS-BPEL code and transformation rules into target WS-BPEL code. Future work concludes the paper.

II. SOA REFERENCE ARCHITECTURE

In order to gain a better perspective of SOA related concepts, it is important to introduce a high-level perspective of SOA reference architecture (Figure 1) [3]. It is consisted of nine layers. Horizontal layers are: Operational Systems Layer, Service Components Layer, Services Layer, Business Processes Layer, and Consumers Layer. Each layer can be described in a logical and a physical aspect. Physical aspect describes how to
implement each layer, whereas logical one describes building blocks of each layer. The following analysis represents logical aspects of layers.

**Operational Systems Layer** represents a collection of applications and data (legacy applications and databases) that already exist in the enterprise. They can be potentially reused and wrapped in order to implement functionalities of components in the **Service Components Layer** [3]. **Service Components Layer** is consisted of code containers that implement functionalities and maintain QoS of exposed services in the **Services Layer**. Wrappers that expose functionalities of applications from the **Operational Systems Layer** are contained within this layer as well. **Services Layer** is consisted of abstract specifications of components from the **Service Components Layer** (WSDL interfaces). Services can be discovered, statically bound, or exploited as a part of composition systems. They are consumed in higher layers, although **Calculation Services**, which perform some intermediate calculation, are consumed in the underlying **Service Components Layer** [3]. **Business Process Layer** is consisted of business processes defined as a composition or coordination of services from the **Services Layer**. Business processes play a central role in bridging the gap between vertical layers:

- **Operational Systems Layer** represents a collection of applications and data (legacy applications and databases) that already exist in the enterprise. They can be potentially reused and wrapped in order to implement functionalities of components in the **Service Components Layer** [3]. **Service Components Layer** is consisted of code containers that implement functionalities and maintain QoS of exposed services in the **Services Layer**. Wrappers that expose functionalities of applications from the **Operational Systems Layer** are contained within this layer as well. **Services Layer** is consisted of abstract specifications of components from the **Service Components Layer** (WSDL interfaces). Services can be discovered, statically bound, or exploited as a part of composition systems. They are consumed in higher layers, although **Calculation Services**, which perform some intermediate calculation, are consumed in the underlying **Service Components Layer** [3]. **Business Process Layer** is consisted of business processes defined as a composition or coordination of services from the **Services Layer**. Business processes play a central role in bridging the gap between vertical layers:

III. **PROTOCOL STACK**

This chapter inspects possible protocol stack of SOA infrastructure. It is important to inspect all standard related issues because standardization is one of the main strengths of SOA paradigm. It has already been stated that services form a backbone of SOA systems and that Web services are their most frequent implementation. Basic protocol stack that implements Web services are WSDL [5], SOAP [6], and UDDI [7] – for description, communication and discovery respectively. WSDL is XML based protocol used to define public interfaces of Web services. Those interfaces describe signatures of offered methods, binding protocols and location of deployment. Offered methods are abstract elements of WSDL interfaces. Web services, whose interfaces share abstract elements, are of the same type. Messages, exchanged between Web services and their clients, are coded using XML based SOAP protocol, and transported using traditional transport protocols, like HTTP. SOAP envelope holds business data (payload), while SOAP headers are used to transport data related to addresses of ultimate receivers. A third protocol, related to Web services stack, is UDDI which represents public or private registries and programmatic interfaces for publishing, retrieving, and managing information about services described therein.

There are mechanisms and protocols for development of composite Web services expressed as a composition or coordination of component Web services. There is a lot of mess when talking of standards used for composition and coordination. Choreography is a coordination mechanism that tracks messages exchanges between component Web services [8]. It describes interactions from a global perspective and it is not meant to be executable. Orchestration models can serve as a blueprint for actual business process implementation [9]. Protocols used for writing choreographies are WS-CDL [10] and WSCI [11]. A comparative analysis of two standards is given in [12]. Orchestration is a mechanism that refers to composition of Web services. The de facto standard, used for specification and execution of orchestrations, is WS-BPEL [13] which has suppressed vendor specific protocols XLANG [14] and WSFL [15]. BPMN [16] is a similar language used for modeling business processes, but BPMN is not executable and it must be translated into WS-BPEL or BPML [17] (although as of 2008 BPML has been reported to have been deprecated).

There is also a family of WS-* specifications that provide a wide range of quality properties for communication processes. **WS-Addressing** [18] provides a standard way of representing addresses of Web services. **WS-Reliable Messaging Framework** [19] ensures guaranteed delivery of SOAP messages and guarantees that messages are delivered in order in which they are sent. **WS-policy** [20] ensures Web service contracts to be extended with policies that express additional constraints, requirements, and qualities that typically relate with the behavior of services. **WS-Security Framework** [21] provides fundamental characteristics of primitive security architecture for XML and Web services like identification, authentication, authorization, integrity and confidentiality. **WS-Discovery** [22] defines a multicast discovery protocol to locate services. **WS-Eventing** [23] and **WS-Notification** [24,25] integrate SOA and EDA architectures. They introduce events to Web services. **WS-Resource Framework** [26] is a family of specifications that allow clients, when talk to the Web service, to include the identifier of the specific resource within the **WS-Addressing** location specification. **WS-Coordination** [27] provides a framework for context management. **WS-Atomic Transaction** [28] enables cross-service transaction functionality comparable to the ACID-compliant transaction features found in most distributed application platforms. **WS-Business Activity** is designed to support long running transactions. It has provisions for such things as exception handling with the capacity to execute compensatory tasks. **WS-Transfer** [29] is a specification used to capture current or intended state of the resource and to transfer that representation between components. **WS-MetaData Exchange** [30] defines different ways metadata may be returned. Those metadata are requested by an endpoint and they must be known in order to effectively communicate with the Web service.

IV. **WEB SERVICE ORCHESTRATIONS**

Mechanisms for integration of component Web services into business processes used to develop composite Web services have already been introduced in the paper. Orchestration is the most accepted mechanism among those. It is tightly related to WS-BPEL language. This paragraph introduces some constructs of WS-BPEL language used to form a skeleton of orchestrations.
Orchestration represents a business process executed in a workflow manner. The process communicates with component Web services via Partners constructs. Partners provide Web services to the process, consume Web services from the process, or interact in a two-way asynchronous conversation with the process. Every WS-BPEL process is itself a Web service described using WSDL protocol. Using this recursive idea as a foundation, more and more complex processes can be created out of lower level Web service relationships.

A main construct that defines relationship between a partner and the process is a PartnerLinkType. It characterizes the conversational relationship between two services by defining the roles played by each of the services in the conversation and specifying the provided portType provided by each service to receive messages within the context of the conversation. PartnerLinkType is defined inside of WSDL interfaces of partner Web services as an extension element using the WSDL 1.1 language extensibility feature. It can be placed into a separate WSDL document when the referenced portTypes are defined by two different services and then the WSDL import element must be used. Another scenario is to place them directly into the originating WSDL document and that is performed when the referenced portTypes exist in the same targetNamespace.

PartnerLinks are instances of PartnerLinkTypes. Within a PartnerLink, the role of the business process itself is indicated by the attribute myRole, and the role of the partner is indicated by the attribute partnerRole. PartnerLinks and PartnerLinkTypes are statically dependent only on the abstract parts of WSDL interfaces but not on actual access information. Actual access information can be dynamically discovered at runtime and obtained using WS-Addressing standard. WS-BPEL process can dynamically select a provider for a particular type of a service. The WS-Addressing standard provides schema for describing access information of chosen WSDL interface.

Communication between the process and its partners can be synchronous or asynchronous. Asynchronous communication is usually applied for long-lasting operations, while synchronous one for operations that return a result in a relatively short time.

Variables are an important constructs used in WS-BPEL processes. They are used to hold incoming messages from partners, outgoing messages to partners, data required to hold the state of a process instance etc. WS-BPEL uses three kinds of variable declarations: WSDL message type, XML schema type (simple or complex) and XML Schema element. WS-BPEL uses XPath as a query language to access variables. Variable updates occur in the assign activity.

During its lifetime a business process instance typically holds one or more conversations with partners involved in its work. So WS-BPEL process needs to deliver messages to the correct instance of the business process that provides the port. When talking about instance management, emphasis is to rely on business data and communication protocol headers and to avoid implementation-specific tokens. CorrelationSet is a WS-BPEL construct used for instance management. It holds a set of properties shared by all messages in the correlated group. The correlated group is a group of operations that use those correlating properties.

Process logic inside of WS-BPEL is expressed using basic and structured activities. Basic activities perform elemental actions while structured ones encode control-flow logic and therefore they can contain other basic and/or structured activities recursively. Basic activities are presented first. invoke activity is used to invoke an operation offered by a partner Web service. Operations can be request-response or one-way corresponding to WSDL 1.1 operation definitions. Business process provides services to its partners through receive, pick and onEvent inbound message activities. reply activity is used to send a response to a request previously accepted through an inbound message activity such as the receive activity. receive activity waits for an invocation to arrive, while receive combined with reply is used to model asynchronous request-response interactions. receive activity plays a significant role in creating instances of the process. The only way to instantiate a business process in WS-BPEL is to annotate a receive activity (or a pick activity) with the createInstanceAttribute set to "yes". The throw activity is used when a business process needs to signal an internal fault explicitly. The wait activity specifies a delay for a certain period of time or until a certain deadline is reached. The empty activity does nothing and it can be used to provide a synchronization point. The exit activity is used to immediately terminate business process instance. The rethrow activity is used in fault handlers to throw the fault they caught. Structured activities prescribe the order in which a collection of activities is executed. WS-BPEL structured activities can be divided into three categories: sequential control, concurrency and synchronization, and choice.

Important construct of WS-BPEL is scope activity. It provides the context which influences the execution behavior of its enclosed activities. scope can contain elements visible only within the scope. Every scope can have handlers attached to it. Fault handler is used to catch a fault whilst compensation handler is used for reversing effects of previous actions. The invocation of a compensation handler can be performed from a fault handler. compensate and compensateScope activities are used to invoke compensation handlers. Handlers are attached to scopes. After a scope starts running, it gives control to its enclosed activities and enables all its event and fault handlers. After a scope completes, it deactivates those handlers. If a scope completes successfully, it enables its compensation handler. When a fault inside of a scope is caught using catch and catchAll constructs, the scope stops execution, disables its enclosed activities, and negatively fires all unevaluated links which sources are in the scope but targets are outside of it. If the scope has a handler for the fault, the activities in the fault handler routine execute. If the fault reaches the process root and does not find a handler, it terminates the entire process. Each fault handler of a scope can initiate the reversal of previous activity results by invoking compensation handlers for its nested scopes. This applies only to nested scopes that have successfully terminated on their normal execution path. Termination handlers provide the ability for scopes to control the semantics of forced termination to some degree. Each scope, including the process scope, can have a set of event handlers. These event handlers can run concurrently and are invoked when the corresponding event occurs.

V. CRITICAL AREAS RELATED TO ORCHESTRATIONS

Although certain issues around writing WS-BPEL code are quite standardized, there are still lots of critical issues that must be better addressed. They are a subject of intensive research.
activities. This chapter identifies some of those areas and gives an overview of existing related research work.

### A. Web Services Granularity Design Pattern

When developing component Web services that communicate with WS-BPEL processes, it is important to determine a right amount of functionality encapsulated inside of them - their level of granularity. Fine-grained services address small units of functionality, or exchange small amounts of data, while coarse-grained services encapsulate larger chunks of capability, reducing the number of requests, necessary to accomplish a task, but on the downside, they might return excessive quantities of data, or it might be difficult to change them to meet new requirements [31]. There must be some well defined criterion that proposes what a well defined granularity is. One of the methods for determining services granularity is top-down functional decomposition. Top-down approach is convenient when a new system is being developed from scratch and it used to identify elementary and reusable service operations [32]. Bottom-up approach is performed when there are existing systems that are going to be wrapped and exposed as Web services or in a process when elementary service operations are aggregated into Web service interfaces [32].

### B. Discovery Mechanisms

Discovery of component Web services is an important issue. There are several approaches for providing service discovery. Dedicated discovery is a very popular approach. Specialized Dedicated Discovery Service performs discovery process based on its internal mechanism. Dedicated Discovery Service is invoked by client services or WS-BPEL process when a process of discovery is needed. It is supposed that it searches for Web services of the same type (i.e. who offer identical methods). After search is performed, Dedicated Discovery Service returns an information (WS-Addressing) referencing chosen Web service and thus allowing dynamical binding of WS-BPEL processes at runtime.

Besides dedicated discovery mechanism, there is a possibility to use extensibility mechanism of WS-BPEL language in order to develop new WS-BPEL constructs for discovery. An approach for service and device discovery developed using special WS-BPEL extensions is presented in [33]. The approach is based on WS-Discovery [22], WS-Transfer [29] and WS-MetadataExchange [30] specifications. For WS-BPEL engines, that do not support mentioned specifications, Dedicated Discovery Services (performing the same logic) are introduced. In general, when introducing extensions to WS-BPEL language, WS-BPEL engines must be adapted to support those extensions and it is a demanding task.

Registries and search engines can be also used for discovery. The most popular Web service registry is UDDI. It supports mainly static Web services with long-term availability. Its shortcoming is a lack of support in commercial applications and a fact that interaction with WS-BPEL processes is not provided directly. Search engines also provide possibility to use extensibility mechanism of WS-BPEL binding of WS-BPEL processes at runtime. Discovery Service processes is not provided directly. Search engines also provide possibility to use extensibility mechanism of WS-BPEL binding of WS-BPEL processes at runtime. Discovery Service

### C. Versioning Design Pattern

Web services versioning pattern includes activities related to keeping information about different versions of Web service public interfaces and to client adaptation to new versions. Supporting different versions of the same service is no built-in feature of UDDI registries, so custom versioning mechanisms must be developed. Reference [36] introduces possible versioning mechanism based on versioning graphs. Adaptation to different versions of Web service interfaces is not performed on client’s side any more. Service registries store all versions of one interface. Service version graph describes how versions relate to each other. Before binding to the service a client calls Invoker Proxy Service. Invoker Proxy Service is initialized with a selection strategy and rebinding interval. After initialization Invoker Proxy Service performs selection over the service version graph and chooses the most appropriate version based on the selection strategy. Invoker Proxy Service periodically requires the service version graph according to the rebinding interval, and checks if the previous binding is still accurate. If there is a new version which better matches criteria according to the selection strategy, Invoker Proxy Service discards the current binding. Only versions with transparent changes can be handled by Invoker Proxy Service. Authors have defined what transparent changes are. Interfaces with non transparent changes have to be proclaimed as new services and if a client wishes to invoke them adaptation on client’s side is necessary. Automatic handling of nontransparent changes is left for future work.

### D. Web Services Invocation

Web services, which communicate with orchestrations, are known as partners. WS-BPEL processes can be statically bound to their partners, or dynamically during the runtime using constructs of WS-Addressing specification. A situation is quite clear when orchestrations communicate with component Web services but there are scenarios when component Web services consume other Web services. This kind of invocation is implemented using invocation frameworks. There are lots of invocation frameworks that automatically generate client proxy stubs based on WSDL interfaces of called Web services. These stubs handle all communication from clients to called Web services but with limitations that they are strongly dependent, not just of abstract parts (offered methods), but of concrete parts of WSDL interfaces as well (bindings, location). If the client invokes a similar service from a different provider, it must regenerate the stub because services offered by different providers in the real world never look quite the same. Even if the services provide similar functionality, they usually differ in technical details. It is obvious that clients configured in this way cannot be considered loosely coupled. It is advisable to develop invocation frameworks that allow looser coupling of Web services and their clients. In [37] Daios invocation framework is developed (Fig 2). It lets application developers create stubless service clients not strongly coupled to a specific service provider. Invocation of Web services using Daios framework is performed in three phases. A first phase is a service discovery phase. In a second phase found WSDL interfaces is compiled in order to obtain necessary information (signatures of exposed methods). Actual service invocation is performed in a third dynamic invocation phase. Daios framework analyzes client’s input and determines to which WSDL input message the provided data best matches according to similarity algorithm. It invokes the operation whose input message has the best structural distance metric to the provided data. The back end is used to conduct the actual invocation to Web services and it is replaceable. Using this framework as a foundation clients are tightly coupled to abstract, not to technical parts of called interfaces. Once a service is successfully bound, any number of invocations can
be realized. Rebinding is necessary if mandatory data change, or if a client explicitly releases the binding.

E. Communication Patterns

This paragraph discusses communication and architectural patterns in Web service compositions. These communication patterns must be well-established. Asynchronous Queuing is a variant of message-queuing systems. It establishes a central queue, in which messages are put and from which messages are consumed, avoiding locking scenarios and allowing consumers to process messages by remaining temporally decoupled. Messages placed onto the queue are stored until the recipient retrieves them. Impacts may come from the facts that there may be no acknowledgement of successful message delivery, and atomic transactions may not be possible. Synchronous Queuing enforces producers to wait for consumers until they take the item and vice versa. Synchronous Queuing can be used for event-triggered communication [38].

Intermediate Routing Pattern provides intelligent agent-based message routing [39]. Message paths can be statically defined during the design time or dynamically computed during the runtime using intermediary routing logic. Routing logic can be computed from the content of message headers or from the content of message bodies and it can be applied on SOAP or even on binary message formats. An impact of this approach is a possibility of performance overhead since dynamical computation of message paths adds layers of processing logic.

Publish/Subscribe Mechanism over extended periods of time is extremely popular communication design pattern since EDA and SOA are tightly related. EDA systems provide an ability to flexibly react to ad-hoc changes, recognize situations, and handle huge amounts of events and data streams [40], whilst SOA systems provide standards compliance, interoperability, and legacy system integration [40]. Since the two architectures have been evolving separately, lots of efforts for filling a gap between them have been introduced. Current state, related to integration of SOA and EDA, is given in [41][42]. Since events play a central role in EDA systems, events and reactions to events must be defined inside of SOA systems as well. A first introduction of events has been done in WS-Eventing [23]. WS-Notification [25,26] is another specification designed for implementation of publish/subscribe mechanism in SOA systems. Direct and brokered are two dominant patterns by which notifications are disseminated in WS-Notification. WS-BaseNotification [24] defines direct pattern where a notification producer is responsible for accepting subscribe messages produced by subscribers and for sending notifications to notification consumers subscribed to them. A notification producer matches notifications to subscriptions and manages subscriptions. WS-BrokeredNotification [25] introduces a notification broker between producers and consumers. Producers send notifications to a notification broker which routes notifications to subscribed consumers. An advantage of WS-BrokeredNotification over WS-BaseNotification is avoiding the need for synchronous communication between the producer and the consumer and relieving the publisher of matching notifications to subscriptions and of managing subscriptions. Publish/subscribe mechanism in WS-notification specification is topic-based. Topics are used as a part of the matching process that determines which consumers should receive a notification. Content-based approach can be also used in WS-notification although topic-based approach is suggested and elaborated in details as a part of WS-notification. An overview of WS-Notification is presented in detail in [43], while a comparative study of WS-Notification and WS-Eventing is presented in [44]. Numerous approaches for implementation of publish/subscribe mechanisms are presented in the remaining part of the paragraph. Most of them are based on extending WS-BrokeredNotification standard.

In [45] WS-BrokeredNotification specification is extended to support registration to complex events and to integrate events and reactions into event-condition-action (ECA) rules (Figure 3). A starting point is the Facade Service which accepts registrations to complex events from complex event consumers. Registrations to complex events combine registrations to atomic events using structural composition operators written using event query language (EQL). Mechanism inside of the Facade Service identifies which EQL is used in the registration and routes it to the correct Event Detector Service. The Event Detector Service splits registrations to complex event into registrations to atomic events. From now on the Event Detector Service acts as an event consumer of all those atomic events. It registers itself at the classic WSN broker and acts as it is interested for all those atomic events. All further communication between the Event Detector Service (acting as Event Consumer), WSN Broker and Event Producers is equivalent to WS-BrokeredNotification. When the Event Detector Service receives notifications about occurrences of all atomic events, it correlates these notifications into a notification about occurrence of the complex event and sends the notification to the Facade Service. The Facade Service forwards the notification to consumers of the complex event. Subscription to complex events can be extended in a form of Event-Condition-Action (ECA) rules. If so, the Invoker Service must be implemented as well. Now notifications about occurrence of the complex event are sent to the Invoker Service which has

![Figure 2: DIAOS Framework for Stubless Web Service Invocation](image)

![Figure 3: Extending WS-Notification With an Expressive Event Notification Broker](image)
previously stored ECA rules. The Invoker Service inspects ECA rules and performs related actions implicitly or explicitly.

In [46] a subscription message contains references to transformations services, restrictions to maximum message size, and a time of delivery (Figure 4). These are all extra features not included in WS-Notification subscription message by default. WS-Notification extensibility mechanism has been used for definition of previously mentioned extensions. Authors in [47] have implemented a distributed content-based notification broker service for WS-Notification. They have implemented content-based publish/subscribe mechanism over distributed hash tables. Authors have also described and evaluated self-optimization mechanisms to reduce the number of notification messages transmitted within the network. An approach in [41] is based on extending WSDL and WS-BPEL specifications.

In [38] integration of SOA and EDA is implemented using Coopetition Services: Queue Service, TokenCenter Service and BrokerCenter Service. Queue Service implements synchronous and asynchronous queuing. TokenCenter Service provides queuing, but in contrast to classical queuing, it allows more than one message to be put or get from the queue. BrokerCenter Service is similar to WSN Broker service. It provides publishing and subscribing with exception that matching announcements to subscriptions is performed inside of Interpreter Services. As part of subscription process, subscribers provide the BrokerCenter Service with the address of the Interpreter service responsible for matching announcements to their subscriptions. Matching announcements to subscriptions by standalone Interpreter Services introduces flexibility. One Interpreter Service can be used for different subscriptions and Interpreter services can be replaced without affecting the BrokerCenter service. Authors have also designed XML based Coopetition Language which uses basic WS-BPEL elements and it is capable to invoke WSDL descriptions of Coopetition Services.

**F. Legacy Systems Integration Pattern**

Legacy systems often offer well established functionalities. In order not to discard them, there are efforts trying to find methods to exploit existing functionalities and to expose them as services. It is important to determine what can be migrated from the original legacy system, and how the migration can be executed. In general, methodologies for legacy systems integration into SOA can be categorized as: black-box reengineering techniques that integrate systems via adaptors, white-box reengineering techniques that require code analysis and modification to obtain the code components to be presented as Web Services, and grey-box techniques that combine black-box and white-box approaches [48].

**VI. MODEL DRIVEN DEVELOPMENT OF ORCHESTRATIONS**

After dealing with critical issues of orchestration development, it is time to deal with WS-BPEL code. WS-BPEL is a language with extremely complex XML markup. Although XML possesses advantages like universal representation and exchange, its syntax is difficult to understand and write for non XML experts. There are lots of efforts trying to develop models in which orchestrations could be written more easily and afterwards would be translated into executable WS-BPEL code. A methodology, when different models and facilitating transformations between those model types are specified, is known as Model Driven Development (MDD). Models may be developed as precursors to implementing physical systems or they may be derived from existing systems or systems in development as an aid to understanding their behavior. OMG has defined a specific set of layers (CIM, PIM, PSM models) and transformations that provide a conceptual framework and vocabulary for MDA [49]. Several languages are proposed for easier development of orchestrations, like: UML, BPMN or WS-CDL.

UML is an excellent candidate because of UML extension mechanisms that can be used for defining so called-domain specific languages. Good tool support is also a reason why to use UML. In [50] orchestrations are modeled using UML4SOA which represents UML2 extended with service specific model elements defined as conservative extensions of UML. UML4SOA introduces a minimal set of stereotypes for modeling SOA orchestrations making UML diagrams less complex. Modeling proposed in [50] follows all rules imposed by OMG. CIM models are presented using UML4SOA activity diagrams. Afterwards they are translated into intermediate PIM models and finally into target PSM models (WS-BPEL and WSDL). A first automated mapping from UML to WS-BPEL is described in [51], but according to [50] it introduces stereotypes for almost all BPEL 1.0 activities – even for those already supported in plain UML, which makes the diagrams hard to read. Similar proposals for transformation from UML to WS-BPEL are presented in [52][53]. Work presented in [52] proposes a development lifecycle consisted of four steps. Preliminary model is constructed in the first phase and it is consisted of an interface of a complex service and of internal behavior of its exposed methods. Interface of a complex service is modeled using UML class diagram while internal behavior of exposed methods is modeled using UML activity diagram. Internal behavior of methods serves as a basis to design WSDL interfaces of component Web services which are later transformed into UML class models. Details from formed WSDL interfaces are added to the composition activity model. At the end composition activity diagram of internal behavior is transformed into WS-BPEL code and interface of a complex service is transformed into corresponding WSDL interface.

Many existing transformations use BPMN for easier development of WS-BPEL code. Its notation is adapted to business analysts as well as to technical personnel. BPMN diagram must be enriched with all implementation details before transforming it into WS-BPEL. Badly constructed BPMN diagram could lead to invalid WS-BPEL code. Work in [54] follows principles of MDD. It presents model driven development of monitored orchestrations using BPMN language. Monitored orchestrations are enriched with sensors which send information to monitoring infrastructure. This information can serve for evaluation of an orchestration.

---

**Figure 4: A Generalized Publish and Subscribe Solution using WS-Notification Standards**
behavior. A complete development life-cycle is shown in Figure 5.

In [55] a lightweight variant of WS-BPEL language (Blite) is designated. Java engine (Blitec) translates Blite code into WS-BPEL code deployable on the engine ActiveBPEL. Blite is a language with simpler syntax and well defined operational semantics. It retains only core features of WS-BPEL because set of constructs in WS-BPEL is not a minimal one [13]. A model written in Blite is consisted of behavioral aspects of the orchestrations and of declarative parts which contain implementation details.

In [40] there is an elaborated methodology (SOEDA) for a complete development of event-driven orchestrations. SOEDA is based on OMG’s principles of model driven development. CIM model is described using EPC diagram. PIM model is expressed as abstract WS-BPEL flow and it is generated using transformation rules from EPC diagram function flow. Another PIM model is a set of event definitions that is generated from event names from EPC diagram using wiring tool. Executable WS-BPEL code and CEP Esper Quer is generated from abstract WS-BPEL adding execution information. CEP Esper Quer is generated from event definitions by mapping event definitions to Esper Quer.

In [56] an approach, based on service composition classes and on composition rules, is presented. Instances of service composition classes are called composition elements, while composition rules are specified in object constraint language. Composition elements and composition rules are designed starting from abstract phase up to construction phase. It is a matter of applying the composition rules to incrementally construct new composition elements and associations during those three phases of composition life-cycle. Finally, resulting information model from the construction phase is translated into WS-BPEL code using transformation rules in the final execution phase.

There are approaches that perform derivation from choreographies into orchestrations [1][2][57][9]. According to these approaches, choreographies serve as starting points and can be seen as models upon which orchestrations are created.

VII. SUMMARY

This paper has introduced main principles of composite Web service development. It has focused on orchestrations as main mechanisms used for this propose. The paper has tried to identify main bottlenecks that must be well defined during development of such compositions. Latter part of the paper has described different models used for easier development of WS-BPEL code since its XML syntax is extremely complex.

Future work will be focused on developing methodology that will propose solutions for identified problematic areas. The idea is to follow principles of MDD so development of an optimal model for easier writing WS-BPEL code will be also in focus of future work.

REFERENCES

[16] Box, D., Christensen, E., Curbera, F., Ferguson, D., Frey, J., Hadley, M., et.al., Web Services Addressing (WS-Addressing), August, 2004., Available at: http://www.w3.org/Submission/ws-addressing/
[18] Bajaj, S., Box, D., Chappell, D., Curbera, F., Daniels, G., Hallam-Baker, P., et.al. Web Services Policy 1.2 - Framework (WS-Policy), W3C


