Semantic Validation of BPEL Fragment Compositions

Marian Harbach, Tim Dörnemann, Ernst Juhnke, Bernd Freisleben
Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str. 6, D-35032 Marburg, Germany
Email: {harbach,doernemt,ejuhnke,freisleb}@informatik.uni-marburg.de

Abstract—We present an approach to improve design-time user support during the composition of web services into BPEL processes. The approach is based on adding semantic service descriptions to SimpleBPEL, a framework that supports web service novices in designing workflows. Experienced BPEL developers define domain specific profiles, containing parts of a BPEL process called SimpleBPEL fragments. The application domain specialists then use these fragments to intuitively create workflows. Semantic service descriptions are leveraged to support workflow design, including data mediation, automated workflow completion and compatibility checks as well as validation based on domain and data semantics. A prototypical implementation demonstrates the feasibility of our approach.

Index Terms—Semantic Validation; BPEL Service Orchestration; Mediation; Semantic Web Services; Semi-automatic Service Composition

I. INTRODUCTION

Web services are a versatile and platform independent technology for accessing functionality in distributed environments. A web service’s interface is described using the Web Service Description Language (WSDL, [1]). Often, a single web service cannot provide the required functionality. The Business Process Execution Language for Web Services (BPEL, [2]) can be used to specify service compositions and create value-added business processes that can be re-exposed as web services, providing more complex operations. BPEL has become the de facto industry standard. Adopting it as a workflow language, in favor of individual and domain-specific solutions, allows scientific applications to benefit from a wide range of industry-grade tools as well as open source products.

Manually writing BPEL compositions is a laborious, error-prone task that significantly benefits from the use of a graphical editor. A significant number of such tools are in use, but they generally target specialist audiences. To provide generic tool support for individual application domains, without the need for creating new languages, Dörnemann et al. have developed DA VO [3], the Domain-Adaptable Visual BPEL Orchestrator. It offers an adaptable data model and allows to customize the user interface so that domain-specific editors for BPEL workflows can easily be built.

Taking this concept one step further, the SimpleBPEL Composer (SBC) [4] abstracts from web services and their invocation patterns. A BPEL expert can define SimpleBPEL fragments (SBF), containing almost arbitrary BPEL activities, and export a collection of SBFs as a domain specific profile. Instead of using surrogate design-oriented languages, like the Business Process Modeling Notation (BPMN, [5]) or UML activity diagrams, the web service novice can load several profiles into the SBC and intuitively compose a workflow with the contained SBFs. Afterwards, the resulting composition can be exported as a regular BPEL process, easily deployed, and reused.

This paper presents an approach to enhanced user support, extending the SimpleBPEL concept: support for composing the SBFs in a valid and meaningful way is provided by leveraging semantic web service descriptions. The aim is to increase ease of use and design-time support for inexperienced users, such that creating a valid workflow is as simple and intuitive as possible. Yet, the added complexity of the formal semantics should remain hidden from the user. In the proposed approach, semantic web service descriptions are used to determine the compatibility of SBF connections, provide support for data mediators to complete a workflow, and validate the completed workflow based on domain and data semantics. A prototypical implementation based on Eclipse demonstrates the feasibility of our approach.

The paper is structured as follows. Section II explores the requirements of employing semantics. Section III outlines our approach, elaborating on the validation strategy, the semantic description formalism, and the mediation method. Implementation details are discussed in Section IV. Section V presents a use case from the medical domain, and Section VI discusses related work. Section VII concludes the paper and outlines areas for future work.

II. BACKGROUND AND PROBLEM STATEMENT

Manually creating BPEL workflows can be a very tedious and error-prone task. End users – who use workflows as a mere tool for developing their applications – might be overwhelmed by the technical complexity of using general-purpose workflow modeling languages such as BPEL. Therefore, we created an abstract view on process creation based on so-called SimpleBPEL fragments (SBFs) [4], containing a combination of BPEL activities. We define a SBF as an arbitrary, but valid, combination of BPEL activities – basically a BPEL process fragment, in combination with a description of input and output connectors to interface with other SBFs.

The BPEL process fragments are composed by experienced BPEL developers who model specific subtasks and informally
describe them in plain text for the end user’s benefit. At this point, the developer also specifies which BPEL variables of the process fragment are in- or output variables for this SBF or whether they remain hidden from the user. In the SBC, the application domain expert is then only concerned with arranging SBFs and creating connections between them, each exposing a set of input and output connectors (the upper and lower semi-circles attached to the rectangles in Fig. 1). The connectors correspond to the in- and output variables specified by the BPEL developer.

Finally, the abstract workflow created by the domain expert is transformed into a regular BPEL process. For this purpose, connections between the SBFs are turned into assign activities in the resulting artifact, providing output values to the connection partner. The activities contained in the SBFs now form subgraphs of the newly created BPEL process. In the simplest case, the contents of an output variable are entirely copied into the connected input variable. However, to be more flexible, message types of the variables involved are not considered to necessarily be a perfect match. Matching the message parts from output to input can only be done based on syntax as well as message or message part names, when using the information available from WSDL descriptions only.

This approach is inherently prone to ambiguity and mostly relies on implicit information, such as that the same message type or message name indicates an actual equivalence of message content semantics. Consider a message named “input” with two parameters, “place” and “temperature” of type string and double, respectively. Furthermore, assume that there is a message from another service named “input” with parameters “location” and “temp” of the same types as above. Even though the first message would be considered as being equal to the second, the first message could refer to a place by a simple name and a temperature in degree Celsius, while the other message could expect a location represented by geographical coordinates in string representation and a temperature in degree Fahrenheit.

To overcome this weakness, we leverage the expressiveness of formal semantics in web service descriptions to make the above information explicit.

Since the problem at hand requires to support the user in composing a workflow at design time, we need to define services and therefore SBFs as pairs of pre- and postconditions to validate the user’s SBF connections. In our context, preconditions describe the necessary conditions on the expected input, while postconditions describe the effect of the element’s execution.

The creation of meaningful semantic descriptions [6] for services itself is not in the scope of our work, hence we assume that semantic descriptions and ontologies for the services are available or can be created by the BPEL developer. Furthermore, the underlying ontologies are assumed to be consistent and aligned, since several approaches exist to achieve alignment of disjunct ontologies [7].

In order to be able to create regular BPEL process descriptions from the SBF composition, the pre- and postcondition describe the input and output message types of an operation. They need to reference the corresponding WSDL elements, allowing to attach meaning to otherwise only syntactically described entities. The SBC is capable of matching individual message parts, and hence the descriptions focus on message parts of the operation’s in- and output messages. While this imposes certain structures onto the description, the semantic description approach needs to allow relationships between message parts and messages to be explicitly detailed.

On top of this, a SBF’s semantic description needs to be derived from the contained activities, since only service invocations will have attached semantics, while other BPEL activities may modify them.

From a pre- and postcondition point of view, we consider only the assign activity to have an effect on semantics. A SBF’s semantics will hence be defined by the semantics of the invocation the exposed variables depend on, taking alterations by assign activities into account.

After having derived SBF descriptions, the tool needs to match pre- to postconditions to provide the user with valid choices for SBF connections. Depending on condition complexity, the cost for computing a match may vary. Furthermore, the larger the number of SBFs in a composition, the more potential targets exist for a connection and the more potential matches have to be computed. It is therefore important to achieve a reasonable response time for user feedback.

Another beneficial mechanism enabled by semantics is the automatic addition of mediators as well as process auto-completion. Concise semantic descriptions allow us to find applicable mediators that overcome heterogeneity on the data level, for example, through conversion from degree Fahrenheit to degree Celsius. Auto-completion is a similar problem, attempting automatic matching of connectors to complete an incomplete SBF workflow.

Eventually, the resulting regular BPEL process can be reused in other workflows. To facilitate integration, the editor
needs to derive a semantic description of in- and outputs of the composite process from the contained semantics.

To sum up, the main goal of our work is design-time composition support for BPEL processes by leveraging semantic service descriptions. The problems to solve include specifying a structure for service pre- and postconditions to match message parts, deriving SBFs from the contained service descriptions, matching the pre- and postconditions in a computationally feasible way to provide validation of compositions, automatically adding mediator SBFs to convert between incompatible variables, providing a means to automatically complete compositions using the semantic descriptions and obtaining a semantic description for the interface of the resulting process.

III. DESIGN OF SEMANTIC FEATURES

To achieve the desired level of user support, the design of the semantic features and their integration into the SBC are based on a trade-off between two requirements: the formal semantics used need to be as flexible as possible to maximize expressiveness, and the user interface (UI) needs to provide near real-time user interaction.

A. Semantic Description

The Web Service Modeling Ontology (WSMO, [8]) and its formal syntax, the Web Service Modeling Language (WSML, [9]), is the most recent formalism applicable to describe web services and their interfaces semantically. WSMO has been used repeatedly in several recent projects (e.g. SWING [10], SUPER [11], and GDI-Grid [12]) and was hence used for this prototypical implementation.

WSML documents for the services involved in SBFs are either available from the service provider or need to be provided by the BPEL developer. Either way, the semantic descriptions need to be linked to the regular WSDL description in the underlying editor’s data model. Therefore, in addition to the WSDL document, the URLs to the corresponding WSML documents have to be provided when a service is introduced into a workflow.

Two kinds of ontologies are expected as the basis for a semantic service description: a data and a domain ontology. This separation stems from the fact that the same domain concept (e.g., temperature) can be represented by different data types (such as double, string or TemperatureType). This approach provides flexibility, as experienced in the geospatial domain [13].

To be able to generate BPEL process descriptions from a composition result with the properties specified above, several structural requirements pertain to the WSML documents, particularly the service descriptions. Listing 1 shows the WSML service description for a sample operation.

First, a WSML web service description document needs to specify the service operation it describes, in order to ground the description with a WSDL document. This is accomplished by the relations anno#wsdlNamespaceRef and anno#wsdlOperationRef in Listing 1, naming the namespace URI, portType and operation described by that WSML document.

Second, due to the structure of messages, every message part needs to be explicitly referenced in pre- and postconditions. Each condition statement is centered around WSML variables that reflect the message parts and specify concept membership or attributes. Furthermore, each part is generally described from a domain point of view and a data point of view, represented by two variables, being members of concepts from both the domain and data ontology. A custom relation, anno#realizedBy(?domainInstance, ?dataInstance), is introduced to indicate that the variable related to the domain ontology is linked to a variable of the data ontology. Semantically, this implies that both refer to the same physical entity, yet leaving the two instances separate, enabling for example, data level type conversion (see Section III-C).

Additionally, the two variables mentioned above need to be explicitly linked to certain message parts of the WSDL document. This is done through another custom relation, anno#wsdlRef(?domainInst, ?dataInst, "msgTypeName/msgPartName"), indicating that the denominated message part’s contents are described by the two instances represented by the variables.

A precondition is treated differently from a postcondition. Preconditions are used as queries while postconditions are instantiated and added to a knowledge base for querying (see Section III-B). The WSDL reference for the input message parts (i.e. input variables) in preconditions is hence attached as an annotation attribute (anno#inputVar2PartMapping) to the capability element, as shown in Listing 1.

To ease the use of the WSML descriptions, we assume that one WSML document describes one service operation only. If a service has more than one operation, multiple WSML documents would describe that service. For simplicity’s sake, we hence use the terms service and operation interchangeably.

Listing 1. An excerpt from a sample semantic description for a temperature conversion service
is just a matter of how to preprocess the service description. We have chosen this approach because otherwise conditions get very complex and need to be decomposed to find the appropriate portion for one operation (see the WSMO Amazon use case as an example of such a WSML description [14]).

The approach shown in the Amazon use case is based on a message level semantic description. Although useful for discovery, message level semantics are usually implicitly obvious (mostly a request/reply structure) from a composition point of view. During composition, the user manually chooses which service to invoke at which point and hence our approach is mainly concerned with matching of message parts. We are aware that this interpretation of WSML documents might be a slight misuse of the WSMO standard. Yet, it facilitates our main goal for this prototype and the issue remains open to be revisited in future work.

We have chosen the WSML-Flight variant for logical expression syntax, since its expressiveness is sufficient to describe service relationships in our use cases but does not allow unsafe rules and function symbols, which allows for efficient reasoning. In addition to the limitations of the WSML-Flight variant, we assume that conditions are specified as a conjunction of terms, since we consider implications and (in-)equalities to be of limited use in pre- and postconditions for composition.

B. Validation Approaches

The validation of workflows in the SBC operates on three levels (cf. Fig. 2): First, a rough pre-validation tests the suitability of a given BPEL process fragment as SBF content. Second, throughout the composition process itself, the online-validation suggests valid target connectors. Finally, the post-validation mechanism accepts complete BPEL processes as input, derives a data flow graph from this input and does a complete validation of the entire graph at once. Since the workflow is incomplete during design time, online-validation simplifies conditions and leaves a full examination to post-validation.

Overall, validation follows a simple assumption: If the precondition of a SBF holds (i.e., all the input connector conditions are satisfiable), then the postconditions (all conditions of the output connectors) of that SBF hold, too. To check whether a given postcondition satisfies a given precondition, the postcondition is instantiated and the precondition is used as a query against the resulting knowledge base.

Instantiation of a condition means that all variables are substituted consistently with valid Internationalized Resource Identifiers (IRIs). The resulting statement is then registered in a knowledge base (KB) as the body of an anonymous WSML axiom. The KB contains the necessary ontologies (data, domain, and annotation ontology), but nothing else. Next, a precondition is used as a query against such a KB, and if a valid result exists, the precondition can be satisfied by this postcondition, because the instances provided by the postconditions fulfill the requirements specified by the precondition. This high-level approach is inspired by the Frozen Facts algorithm for query containment [15]. Hence, the same limitations apply: the conditions need to be positive and cannot contain built-ins or disjunctions.

1) Pre-Validation: Pre-validation checks whether BPEL process fragments are consistent before they are accepted as SBF contents. This ensures that they can be properly applied in the domain user’s view. Validity checks include absence of receive or reply elements as well as restrictions on copy elements in assign activities. The latter is vital for deriving SBF semantics: If the copy element contained complex from specifications (e.g., complex XPath expressions), the semantics of the variable given in to would have to be manually specified, due to the absence of automatic reasoning with arbitrary XPath expressions. Hence, we allow direct copying of variable contents and literal values only.

At this stage, the system also needs to derive the SBF’s semantic description from the contained operation descriptions. Therefore, the origins of the SBF’s in- and output variables are examined: If they are directly originating from an invoke activity, the corresponding semantics are assumed. Otherwise, due to the constraints stated above, the variable contains either constant values or parts copied from various other variables. Originally, the variable’s contents relate to either a service input or output, and by finding the corresponding invoke activity, we can gather the required part’s semantics.

The result of pre-validation is a SBF description, with input and output connectors for the corresponding in- and output variables as described above. Additionally, the connectors have a condition attached, which is derived from the fragment contained in the SBF.

2) Online-Validation: User support in composing a workflow is mainly provided by the online-validation mechanism. While the domain user attempts to connect SBFS, the system indicates compatible variables and refuses connections to incompatible ones, as shown in Fig. 3. The same mechanism can be applied to auto-complete an incomplete composition. The compatibility calculation is one of the main features in
the original SBC and can be easily extended via an Eclipse extension point.

Fig. 3. A sample workflow with active online-validation: While the user drags the mouse to make a connection, the system calculates and indicates appropriate connection targets (the non-filled semi-circles).

The semantic extension can leverage the SBF’s semantic descriptions for compatibility checks. It uses the condition of the potential target connector as precondition and the condition of the source connector as postcondition. A match is then attempted using the instantiation method described above. If a full match for the entire precondition cannot be obtained, the system splits the precondition by message parts and tries to match individual parts. If a valid binding is obtained either through a full or partial match, we can extract which message part needs to be copied into which message part from the anno#wsdlRef and anno#inputVar2PartMapping information of the connector conditions. This mapping from source to target parts is saved and used for generating assign activities in the resulting BPEL process. When all message parts of one input connector are mapped, it is considered to be complete, and if all of its input connectors are complete, the SBF is considered to be complete.

An immediate problem arises for online-validation when service descriptions use shared variables. Shared variables allow postcondition variables to be bound from the bindings of the corresponding precondition. When this happens, the postcondition of a service depends on the semantics and binding results of a chain of predecessor services. While deriving SBF semantics in pre-validation and when the composition is still incomplete during design-time, the predecessors of a service remain unknown, hence shared variables have to be ignored. This leads to problems when trying to bind preconditions: While a postcondition might satisfy a precondition when shared variables are taken into account, it might not do so otherwise. This issue stems from expressions in preconditions that reference attributes pertaining to instances that are not instantiated by the immediate predecessor but are supposed to be “inherited” from services upstream from that neighbor via shared variables.

We use two methods to overcome this limitation for online-validation: In pre-validation, the SBF preconditions are truncated by variable depth. In online-validation, shared variables are ignored and postconditions are instantiated using new concept identifiers. To truncate conditions, each variable is inserted into a dependency tree, starting from the variables mentioned in the anno#realizedBy relation. The children of a node are the variables that are mentioned as attributes of the current variable. After normalizing the condition, we remove all terms from the conjunction that contain a variable with a dependency depth beyond a specified maximum. The maximum depth is 1 for the most general case, since otherwise a term could reference a certain property of an attribute that is not specified by the direct neighbor service. For example, in a data conversion scenario, a precondition could be ?input[convertedFrom hasValue ?input2] memberOf aDomainConcept and ?input2 [originalFile hasValue ?input3] and ?input3 memberOf someDomainConcept. In this case, ?input3 has depth 2 if we consider ?input to be the root of the dependency tree. Hence, any statement involving ?input3 is removed from the conjunction. The remainder would be ?input[convertedFrom hasValue ?input2] memberOf aDomainConcept, which would allow us to bind the immediate neighbor SBF at design-time without breaking the semantics, because ?input2 can now match any instance. These dependencies can be fully validated in the post-validation step.

3) Post-Validation: Once a SimpleBPEL process is complete (i.e., all SBFs are complete), post-validation is initiated. The process’ interface semantics are then derived from the connectors connecting to the Start and Stop SBF of the SimpleBPEL composition. Post-validation can be applied to any BPEL process containing semantic descriptions that conform to the requirements specified in the pre-validation section. In contrast to online-validation, post-validation validates a complete process graph. This allows the validation to take shared variables and hence chained dependencies of conditions into account.

In a first step, the BPEL process is converted into a data flow representation [16], abstracting from the complex process description as a directed graph. In this graph, invoke activities are the nodes and an edge is drawn between two nodes if there is an assign activity, copying values from the output variable of one invoke to the input variable of the other. This corresponds directly to the dependencies of a service’s precondition in terms of the postconditions it can bind its variables from.

The validation itself is a breadth first traversal of the data flow graph. The algorithm keeps a knowledge base (KB) as context. Initially, all the ontologies referenced in the process are added. Then, the precondition of the process itself is instantiated, since it holds when the process is invoked from the outside. Afterwards, the graph is traversed in a breadth first manner, and the preconditions of each service operation (equivalent to an invoke element and a data flow graph node) are bound from the context KB. Using this binding, the postcondition is instantiated with respect to any shared variables and then added to the KB. If a precondition cannot be bound, the validation returns an error.

When binding preconditions from the context KB, ambiguities can occur if one service is contained multiple
times in the data flow graph or if two services happen to have similar postconditions. To prevent this, each condition instantiation is marked with a unique service instance identifier. This ID is attached to the instantiated condition before adding it to the context KB. A ternary relation, `anno#providedBy(?domainInstance, ?dataInstance, 'uniqueID')`, is appended for every `anno#realizedBy(?domainInst, ?dataInst)` term. When attempting to bind a precondition, the query is extended to include the `anno#providedBy` relation with a variable instead of the uniqueID. The reasoner returns a set of bindings for each query, and the system can then use the IDs returned in the query to remove bindings that do not come from a set of IDs specified when attempting the bind.

C. Mediation

In various contexts, the term mediation is used for different aspects when talking about web service interactions. The WSMO standard [8] generally distinguishes between four kinds of mediators: A Goal Mediator mediates between two goal descriptions, stating, for example, that one goal is a refinement of the other; Ontology Mediators are concerned with resolving heterogeneity between two ontologies; Service-to-Goal Mediators link web services and orchestrations to goals, in terms of required and fulfilled goals; and Web Service Mediators resolve “any data, process and protocol heterogeneity” [9] between two web services.

The mediation implemented in the SBC realizes a part of web service mediation. As mentioned above, the main goal in our approach is user support for web service composition. Furthermore, ontologies are assumed to be already aligned and goals are implicitly given by the (partial) composition created by the user. Hence, there is only a need for service level mediation, and since SimpleBPEL is not concerned with orchestration aspects of services, the only remaining mediation issue is at the data level.

SimpleBPEL mediators enable the user to connect two SBFs with incompatible conditions, if and only if there is a suitable mediator registered with the system. These mediators are created by a BPEL developer as regular SBFs and explicitly marked as mediators on profile export. When a direct match between in- and output connectors cannot be obtained, the system evaluates all SBFs marked as mediators. If the mediator’s precondition matches the source SBF’s postcondition and the target SBF’s precondition matches the mediator’s postcondition, then the SBF is automatically added into the composition and the necessary connections are established. While visually adding such an extra SBF to the process might irritate the user, its outputs can be reused to connect to other SBFs. Therefore, it is beneficial to visualize the mediator SBFs, and the additional value outweighs the potential irritation.

To realize this approach at design time, SBFs intended to become mediator SBFs can only expose one input and one output variable. Additionally, conditions are only matched entirely and not partially as for direct connections. Otherwise, evaluating the applicability of a mediator SBF would be computationally too expensive for design-time support. For each possible target, all the mediators have to be evaluated to check whether they can facilitate a match or not. Obviously, with an increasing amount of SBFs and/or increasing complexity of conditions, more time is needed to compute these matches. To reduce latency, a restriction on the number of variables has been chosen.

However, this approach to mediation comes at a certain price. Above, we assumed to ignore shared variables for online-validation. But for mediators with very generic conditions, this restriction leads to poor user support. Consider a workflow with rather complex conditions for the main SBFs. Additionally, suppose we have some data value that represents some domain concept, e.g. temperature. Imagine we want to match two SBFs, and the only incompatibility is that in the source SBF, temperature is provided as a double value, while the target SBF expects temperature as an integer. One could come up with a simple mediator SBF: its conditions would basically state that it takes whatever domain concept as input – as long as it is realized by a number – and returns that very same domain concept realized by an integer. To keep this mediator generic, we need to use a shared variable to ensure that the domain concept instance in the postcondition is the very same as the one in the precondition. When applying the limitations described above, this type of mediator would not provide a match due to the lack of shared variable consideration.

To overcome the shared variables limitation, we added so-called simple function SBFs (SFFs). A SFF can only contain a single `invoke` activity and can hence directly expose the underlying service’s pre- and postcondition as well as the associated shared variables. Since there cannot be any dependency problems in the contained BPEL process fragment, the limitations described in the pre-validation section no longer apply. Hence, we extended our online-validation approach to consider the shared variables of SFFs. To apply shared variables, we need to recursively bind all postconditions of SBFs connected upstream before binding the actual precondition of the SFF and using that binding result to instantiate the postcondition. This step is called upstream instantiation and stops when a non-SFF is found. The task is computationally feasible for design-time support, since the limitations of the BPEL process fragments in SFFs allow only one in- and output variable per SFF, hence creating linear chains only. Furthermore, we generally expect rather simple conditions and only short chains of SFFs.

IV. Implementation

To implement our approach, the SBC has been extended with a prototypical implementation of the semantic extensions. The Eclipse RCP plug-in for DAVO and SimpleBPEL have been modified to accommodate semantic descriptions for the relevant elements. Furthermore, some changes have been made to incorporate `MediatorFragments` and `SimpleFunctionFragments` in the SimpleBPEL model. The profile creation view for DAVO has been extended to allow selection of SBFs for mediation purposes and to check the derived semantic description for the SBF connectors.
V. Evaluation

A real-life workflow originating from the area of sleep research has been used to evaluate our approach. The workflow basically performs an ECG (electrocardiogram) analysis and, based on the obtained results, conducts apnoea detection. The implementation uses the Physio Toolkit [20], a common set of open source tools in biomedical sciences.

Since the data format (European Data Format, EDF) of the recorded vital signs (real, anonymized patient data) is different from the format required by the Physio Toolkit (WaveForm DataBase, WFDB), a data conversion is needed (invokeEDF). Afterwards, Q-S peaks are detected within the ECG signal (invokeWQRS). The results are passed to the annotation reader service (invokeAnnotationReader) that in turn decodes the input and passes the results to the beat detection service (invokeBeatDetection) that detects R waves within the signal. In parallel, the output of invokeWQRS is passed to the apnoea detection service (invokeApnoea) that analyzes the input signal and detects respiration dropouts (to diagnose the sleep apnoea syndrome). As described, the original workflow consists of five service invocations, but in the SimpleBPEL abstraction, only four SBFs are visible (see Figure 1).

All services were semantically described using a sample ontology for describing ECG data as well as a simple data level ontology. All the relevant files including the Eclipse plug-in and a set of instructions to recreate this use case are available at http://image.uni-marburg.de/trac/gdt/wiki/SimpleBPELOverview.

After loading the SBFs, the annoBeat SBF is modified to additionally expose the annoInputVar as input connector. For this SBF, the variable beatInputVar2 has been created in addition to the default input and output variables, to expose some of the actual message parts of the beatInputVar variable only. The system can detect this correlation and accordingly extract a valid semantic description for beatInputVar2 from beatInputVar (which refers to the precondition of the beatDetection service description), using only the part concerning the ECG data reference. At profile export, invokeEDF, invokeWQRS and invokeApnoea are registered as possible mediators, since they only have one input and output variable each.

After profile export, a new SimpleBPEL process is created in the SBC. Next, the Start and Stop SBFs are dragged into the modeling canvas and the user starts the process creation with an invokeEDF SBF. This is necessary because we cannot know the semantics of the input and output variables for the overall process beforehand, which hence have to be inherited from the SBFs that are initially connected to the Start/Stop SBFs. Next, let us assume the user knows that (s)he wants to use the invokeAnnoBeat and the invokeApnoe SBF, because these SBFs compute the information the user wants to obtain.

When attempting a connection with the legacy syntactic compatibility modes, no matches are found for most of the connectors, since invokeApnoea and invokeAnnoBeat both expect the result of the invokeWQRS service. Some matches are possible, but semantically incorrect, because, by coincidence, the input messages of invokeApnoea and invokeAnnoBeat are syntactically equivalent to the invokeEDF input message, i.e. available for syntactical matching from the Start SBF.

Using the “Simple Semantics” compatibility mode, only the correct matches are indicated. When the user tries to connect invokeEDF to invokeAnnoBeat or invokeApnoe, the system automatically adds the invokeWQRS SBF as a mediator and creates the necessary connections. After connecting the intended output variables to the Stop SBF, the process is complete and post-validation does an overall check of the resulting BPEL process. The completed and validated process can now be exported and deployed to a workflow engine or post-processed with any BPEL editor.

VI. Related Work

Several approaches to web service composition have been described in the literature. Laga et al. [21] have surveyed the available approaches for service composition and classified them into three categories: automatic, semi-automatic and static composition. Automatic composition approaches accept a user goal as input, expressed in natural or formal language, to discover and automatically compose suitable services to reach that goal. This functionality is implemented in WSMX [22], for example. Despite a significant amount of research into fully automatic composition, Laga et al. have identified a need for user-centric approaches. Semi-automatic composition is presented as a means for a user to drag-and-drop objects and connect them to achieve some new functionality. This approach is supposed to target non-expert users, in contrast to static composition that requires programming skills. While the SBC presented in this paper fits with their definition of a semi-automatic composition tool, its features go beyond the capabilities of other approaches by providing semantic validation using SBFs and integration with enterprise-style execution environments.

Chafle et al. [23] have detailed the need for Integrated Development Environments (IDEs) in the context of web service compositions. They argue that adequate tools are required to decrease development time and facilitate integration. They present Synth, an IDE to provide end-to-end support for service composition. In contrast to our approach, their product is tailored towards workflow experts and does not offer the simplicity inexperienced users need.

In a similar way, the SUPER project [11] provides an IDE for end-to-end support of semantic web service compositions.
to business process modeling specialists. They leverage semantic descriptions to create an environment that supports the modeling specialist without the need for IT experts. However, as with Synthy above, this comprehensive approach is far from begin simple and does not hide complexity from novice users. It is also not tailored towards use in legacy workflow environments, but rather rests upon a semantic web service infrastructure.

Early efforts in service composition mainly include agent-based approaches (for example, [24]), that try to enable an agent to achieve a goal by interacting with other agents. The use of semantics is required to let agents understand each other’s goals and services. According to Hull and Su [25], artificial intelligence planning problems have a similar structure to service composition goals, and it is hence not surprising to find many interrelated concepts in the two domains. The MINDSWAP group has presented a semantics-enabled prototype [26] for service composition with DAML-S and WSDL before the standardization and wide use of BPEL4WS emerged.

While most related efforts achieve comparable flexibility in general semantic support for service composition, to the best of our knowledge, none of them focuses on inexperienced users explicitly. The ability to create SBFs from BPEL process fragments, allowing application domain experts to compose them and creating an easily deployable process description for enterprise style workflow systems remain unique properties of our approach.

VII. CONCLUSION

We have presented an approach to semantic validation of BPEL fragment compositions. The design-time user support for BPEL workflow composition was improved by extending our SimpleBPEL Composer. An abstraction from plain BPEL processes based on BPEL process fragments has been proposed. Formal semantics have been used to determine compatibility for SBF connections. In contrast to other approaches, we have focused on design-time support for end users and integration with enterprise-style BPEL workflow systems.

Limitations pertaining to the semantic descriptions, stemming from computational complexity, have been tackled by a three-tier validation approach. The pre-validation step prepares BPEL process fragments for their use in our modeler. The online-validation then indicates compatible objects to the user at design time and suggests data-level mediators to complete the workflow. After the workflow is completed, a standard-compliant BPEL process is created and a semantic description for the new process is automatically derived. The post-validation step finally processes any semantically annotated process and validates its overall structure, including dependencies that may exist among several service invocations.

Future work includes a complex mediation scenario that allows arbitrarily complex mediator combinations, resembling a fully automatic composition. Additionally, data level mediators could be hidden from the user and an optimal configuration could be automatically created by the system. Support for the WSML-DL variant would furthermore allow the software to process service descriptions based on description logic. Finally, a tighter coupling with a reasoner could reduce overhead and enable more comprehensive online-validation methods.

VIII. ACKNOWLEDGEMENT

This work is partly supported by the German Ministry of Education and Research (BMBF) (D-Grid Initiative).

REFERENCES