Semantic Service Specification for V&V of Service Composition and Business Processes

Ralph Hoch, Hermann Kaindl, Roman Popp, Dominik Ertl
Vienna University of Technology, Vienna, Austria
{hoch, kaindl, popp, ertl}@ict.tuwien.ac.at

Helmut Horacek
German Research Center for Artificial Intelligence, Saarbrücken, Germany
helmut.horacek@dfki.de

Abstract

Semantic specification of services based on formal logic can be used for automated verification of service composition. In order to make such verifications consistent with validations of service compositions in the context of business processes, more and more knowledge needs to be included in the related specifications. We show using a simple example that after adding such additional knowledge directly to the semantic specifications of services, they may become over-specified. We found that this additional knowledge can be a special kind of business rules. Therefore, we propose to specify them separately, but also based on formal logic. More precisely, the use of the Fluent Calculus and the related FLUX tool enabled automated and guaranteed verification of composed services against the specifications of the single services. Adding the formalized business rules into such verifications made them consistent with validations of service compositions in the context of business processes. Overall, both verification and validation (V&V) are essential for service composition and business processes. As a consequence, this novel approach to V&V should support a comprehensive approach to service design.

1 Introduction

Service composition is the process of creating a more complex (Web) service from other (Web) services. If such a service does not rely on any other services for its functionality, it is considered a simple or atomic service. Otherwise, we refer to it as a composite service [1, 5]. Examples could be services for creating, authorizing and sending invoices, which are standalone/atomic. They form the new composite service “Issuing Invoice” through service composition.

Much as any software, composite services and (business) software composed from services needs to be tested. This can (hopefully) find bugs and design problems but not provide any real guarantees. In contrast, semantic service specification based on formal logic allows for formal verification of composed services against the specifications of the single services. More precisely, our verification approach checks whether a defined sequence of service invocations is consistent with the semantic specifications of the services involved. In this sense, our verification means to formally and automatically check whether the composition is built right based on the single services.

In this context, we pose the question whether semantic service specification is sufficient for such a verification of composed services for implementing business processes. Verification based on logic (involving formally specified pre- and postconditions) may reveal loop-holes in the knowledge represented in the specification of a composed service with regard to the business process. Even though the verification of a composed service may be successful, its validation as a (fragment of) a business process may not. That is, the composition is not right as a business process. In such a situation, adding missing knowledge to the service specification may create a mismatch between this specification and the service implementation. So, we added specific business rules and achieved consistent results from V&V of service composition and business processes.

Our methodological approach to answering this question is to formulate the hypothesis that semantic service specification alone is sufficient for such a verification of composed services for implementing business processes. By providing an example where this is not the case, we can reject this hypothesis. For providing such an example, we use very simple business processes for a small and a large company each (both just hypothetical for the purposes of this paper), including the tasks Create Invoice, Authorize Invoice and Send Invoice, as well as their respective dependencies. For
each of these tasks we assume implementations as Web services, which are to be composed accordingly. This example is very simple on purpose. We claim that the issue shown with it will most likely be relevant for any real-world processes as well, since it even occurs in this simple example.

As a means to this ends, we employ given theories and their supporting technology. When using such knowledge represented in the Fluent Calculus [21], verification of a composed service can be done in the related tool FLUX.\footnote{FLUX agent: http://www.fluxagent.org/} It has fully defined semantics also of its reasoner, which can be employed for this verification. This entails the necessity to specify pre- and postconditions in the FLUX tool.

Still, we had to extend this approach for our purposes of V&V. We did this by additionally representing and including a certain kind of business rules here.

We consider V&V of services an integral part of service design. Our approach may serve as a holistic approach, since it integrates V&V of services and business processes.

The remainder of this paper is organized in the following manner. First, we present some background material in order to make this paper self-contained, and discuss related work. Then we specify semantic knowledge for service composition using our example and uncover an inherent problem. Based on that, we propose a solution by making business rules explicit for their use in the verification and validation (V&V) of service composition and business processes. Finally, we discuss our approach more generally, indicate future work and conclude.

## 2 How to Use FLUX for Formal Verification

While the Fluent Calculus is just a means to our ends of V&V, this calculus is the basis of our V&V approach. More precisely, it allows for formal verification through its related tool FLUX. So, it is necessary to provide here some background material. In this course, we focus on the use of FLUX for formal verification of composed services against the specifications of the single services.

We chose an implementation of FLUX built upon the constraint logic programming system ECLiPSe since our aim is to compose services in a certain way based on constraints.\footnote{ECLiPSe Constraint Programming System: http://www.eclipseclp.org} The Fluent Calculus as supported by the FLUX tool is already built for a constraint programming language and thus the formal specification of the Fluent Calculus does not have to be implemented again [22].

The Fluent Calculus provides a formalism to model specific operations that transfer from one state to another. This is specified using the poss and state\_update predicates. These predefined predicates are used to model the preconditions (poss statement) and postconditions (state\_update).

Together, they provide a formal specification of an operation.

To illustrate how such an operation and its effects can be modeled in the Fluent Calculus, we use a simple example. Let us assume that there is an operation which creates, under certain conditions, an invoice. An address and an amount are input, and a corresponding invoice is output. In addition, there is a constraint on the invocation of this operation, that no such invoice exists yet. Strictly speaking, several other invoices may exist, so that the condition would have to be more specific to the given business case (indicated here through the address and amount), but for reasons of simplicity of the example we omit this here. Listing 1 provides a semi-formal overview of this CreateInvoice operation.

### CreateInvoice:

<table>
<thead>
<tr>
<th>Input</th>
<th>Signature</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address, Amount</td>
<td>createInvoiceOperation(Address, Amount)</td>
<td>Invoice</td>
</tr>
</tbody>
</table>

**Precondition:** exists(invoice, false)

**Postcondition:** none

Listing 1: CreateInvoice Specification

Now let us construct a corresponding formulation in FLUX. Listing 2 shows the signature of the operation createInvoice with its two arguments Address and Amount, createInvoiceOperation(Address, Amount). This listing also shows how the poss and state\_update predicates can be connected in FLUX, where Z denotes the current state and Z2 the state resulting from the state\_update.

\[
\text{poss}(\text{createInvoiceOperation} \ (\text{Address}, \ \text{Amount}), \ \text{Z}) , \\
\text{state}_{-}\text{update}(\text{Z}, \ \\ \\
\text{createInvoiceOperation}(\text{Address}, \ \text{Amount}), \\
\text{Z2}, \ [])
\]

Listing 2: Signature of CreateInvoice operation in FLUX

Still, it is necessary to provide a formal specification of the pre- and postconditions. It is shown in Listing 3, where predicates predefined in the Fluent Calculus specify checks of the input. The first part is the head of the predicate poss. It takes two arguments, the signature and the current state. This head is separated from the body through the \(-\) delimiter. The body consists of three constraints modeling input and precondition, which are logically connected with and each. Checking input is specified with the predicate knows\_val and checking preconditions with the predicate knows\_not. knows\_val specifies a check whether for a given variable a value can be found in the current state. knows\_not specifies a check whether a given value is not known in the current state. There is an additional predicate holds, not used in this example so far, for specifying whether a given predicate or literate holds, i.e., is known to be true, in the current state Z.

\[
\text{poss}(\text{createInvoiceOperation} \ (\text{Address}, \ \text{Amount}), \ \\
\text{Z}) :-
\]

1^FLUX agent: http://www.fluxagent.org/

2^ECLiPSe Constraint Programming System: http://www.eclipseclp.org

1371
knows_val([Address], string(Address), Z), % Input %
knows_val([Amount], float(Amount), Z), %
Input %
knows_not(invoice(_), Z). % Precondition %

Listing 3: CreateInvoice-Operation preconditions encoded in FLUX

Since Listing 3 only shows conditions that need to be fulfilled before the invocation of the operation, a second part specifying the updates is necessary. Listing 4 shows how postconditions can be modeled in FLUX. Again, the signature createInvoiceOperation(Address, Amount) is used to identify the operation, and an update statement models state transfer. In general, update takes several arguments, where the second argument specifies the statements to be added to the new state and the third argument the statements to be removed from the previous state. These two arguments specify the add and delete list of predicates that are applied to the current state Z1 and form the new state Z2. This mechanism allows new predicates to be introduced as well as existing predicates to be removed from a state, in contrast to, e.g., predicate calculus. In effect, predicates or facts can change over time or more precisely after invocation of an operation. Furthermore, this enables the calculus to negate existing facts or set negated facts to true. In this example, the second argument specifies that invoice(Address, Amount) is to be added to the new state Z2, while the third argument is empty since nothing is to be removed.

\[
\text{state_update}(Z1, \text{createInvoiceOperation(Address, Amount), Z2} : [ ]) :- \\
\text{update}(Z1, [\text{invoice(Address, Amount)}], [], Z2). \ % \text{Output %}
\]

Listing 4: CreateInvoice-Operation postconditions encoded in FLUX

Such specifications provide the basis for a formal verification using the FLUX tool. For illustrating this, we introduce yet another operation for sending invoices. Listing 5 shows how such an operation could be defined.

SendInvoice:
Input: Invoice
Output: none
Precondition: none
Postcondition: sent (Invoice)

Listing 5: SendInvoice Specification

The FLUX specification is created analogously to the one for the operation CreateInvoice. In order to show such a specification in one piece, we include here Listing 6.

\[
\text{state_update}(Z1, \text{sendInvoiceOperation(Invoice), Z2, []} : [ ]) :- \\
\text{update}(Z1, [\text{isSent(Invoice, true)}], [], Z2). \ % \text{Postcondition %}
\]

Listing 6: SendInvoice encoded in FLUX

With these two specifications of operations in place, we can perform a formal verification of, e.g., sequences of operations using the FLUX tool (for our purposes of composed services or sequential business processes). For example, in the simple business process shown in Figure 1, first an invoice is created and then sent out.

![Figure 1: “Create and Send” Business Process](image)

This abstract definition of such a sequential business process can be specified in FLUX as shown in Listing 7. The first statement Z = [string(“ExampleAddress”), float(“55”)] is just an additional statement to specify an initial state. In this case, just two facts of the form string(“ExampleAddress”) and float(“55”) are introduced for use in the operation CreateInvoice. After this initialization, the poss statement specifies a check whether the first operation CreateInvoice can be invoked. If poss evaluates to true, then the state_update is performed. The third step specifies a check whether the operation SendInvoice can be applied and, if yes, the state_update is performed. The FLUX tool interprets all these specifications based on well-defined semantics and tells that this verification succeeds.

\[
Z = [\text{string(“ExampleAddress”), float(“55”)},]
% \text{first check if the operation \text{CreateInvoice} is applicable %}
\text{poss(createInvoiceOperation(Address, Amount), Z),}
% \text{then perform the operation through \text{state_update %}
\text{state_update(Z, createInvoiceOperation(Address, Amount), Z2, []),}
\% \text{check if \text{SendInvoice} is applicable %}
\text{poss(sendInvoiceOperation(Invoice), Z2),}
% \text{and then perform its update %}
\text{state_update(Z2, sendInvoiceOperation(Invoice), Z3, []).}
\]

Listing 7: “Create and Send” Business Process encoded in FLUX

As shown in this example, this sequence of poss statements along with their corresponding state_update statements has to be provided to the FLUX tool for specifying the verification of the sequence defined in the business process shown in Figure 1. If all the given statements can be
performed in this order, the verification succeeds, otherwise it fails. So, this is a verification of whether a defined sequence of service invocations is consistent with the semantic specifications of the services involved.

In FLUX, service specifications are always evaluated on the current state. That is, all information that is present is also available to the services. This enables the possibility for more complex links between services. For example, once CreateInvoice has provided an invoice, SendInvoice may use it even though other services are invoked in between, unless any of them explicitly deletes this invoice.

Of course, the human user of FLUX is responsible for the \textit{state_update} operation and has to take care of conflicting statements. For this example, the \textit{state_update} operation must not only introduce the new statement \textit{sent(Invoice)} but also remove the statement \textit{~sent(Invoice)} from the new state.

3 Related Work

Salomie et al. [19] studied Web service composition using the Fluent Calculus, viewing automatically composing Services as an Artificial Intelligence planning problem. This service composition technique has been further discussed by Bhuvaneswari et al. [4]. While the techniques for automatically composing services and for verifying a given composition are closely related and both supported by FLUX, none of this previous work addresses our main topic — V&V of service composition and business processes.

Another approach for semantic specification of services has been developed by Baryannis et al. [3], where an intermediate language (WSSL) is formulated. WSSL enriches the basic and standard service specification in WSDL with pre- and postconditions. These are actually based on the Fluent Calculus and could, in principle, be used for our verification purposes as well. In fact, our formulations of pre- and postconditions in FLUX as given above were informed by WSSL.

An earlier approach is based upon the Web Ontology Language (OWL), which is a knowledge representation language used to build and administer ontologies or a specific knowledge base.\footnote{OWL Web Ontology Language: http://www.w3.org/TR/owl2-overview/} OWL-S is an ontology built upon OWL for semantic descriptions of Web services. It aims at automatic discovery, invocation and composition via autonomous software agents. OWL-S consists of three parts: Service Profile, Process Model, and Service Grounding. The latter provides the means for interoperability with a Web Service Description (given in WSDL) and relates the semantic specification of a Web service with its WSDL file. This involves the definition of the input and output parameters including their types. In addition, OWL-S provides pre-defined predicates for defining preconditions, result values and effects. Services can be modeled either as atomic or as more complex composite services, where the latter consist of several (orchestrated) atomic services.

There have already been approaches for automatic planning algorithms based upon the OWL-S specification of services which try to automatically generate composite services out of atomic services, e.g., Klusch et al. [12] and Ziaka et al. [27]. However, the approach based on the Fluent Calculus and FLUX seems to be preferable because of its well-defined semantics.

With respect to the real ends of our own approach built on FLUX, none of these approaches deals with additional knowledge required for the service composition that should \textit{not} be embedded into the service specification itself [11].

Related research on implicit business knowledge (tacit knowledge) is still in its infancy. Chesbrough et al. [6] emphasized in their research manifesto for services science that the nature of tacit knowledge complicates the services exchange — and service exchange is a corner stone for service composition. More precisely, tacit knowledge limits the ability of each service-party to fully comprehend the needs and abilities of each other. The authors point out that a multidisciplinary perspective toward services becomes increasingly important to gradually codify tacit knowledge. We actually found very simple cases where tacit knowledge needs to be made explicit for V&V of service composition and business processes, and we show that this is business knowledge that should not be encoded in the service specifications.

Montali et al. [13] introduced an alternative way of specifying service choreographies by directly defining them through a set of policies referred to as constraints. These are embedded in their flow language, explicitly connecting services, e.g., with their time flow. Their approach uses Linear Temporal Logic to verify conformance checking, conflicts and dead activities, interoperability between global and local models, etc. In contrast, we focus less on the choreography aspect but explicitly represent business rules declaratively and use the Fluent Calculus for verification of composed services against the specifications of the single services including business rules. These are atomic entities in our approach like services, while Montali et al. embed them directly in the flow.

Feng et al. [8] proposed an approach for verifying properties of the process model of an OWL-S service. Via mapping rules this approach translates the process model into a process algebra model and uses a model checker to verify the properties of such a translated model. It handles the control flow as well as the binding-based data flow of the process model. In contrast to our approach, implicit knowledge is not separated from the service specification. Our work makes the (tacit) business rules explicit that actually
glue services together semantically in a business process.

Li et al. [26] used Propositional Logic for requirements verification of service workflow, where requirements include business rules. Their work is capable of checking compliance and also of detecting conflicts of imposed requirements. However, its focus is mainly on compliance checking between a Service Workflow Net (SWN) and SWSpec formulas. This is different from our research on whether semantic service specification is sufficient for V&V of service composition and business processes.

Ni et al. [14] transformed models and formally verified semantic Web services composition. More precisely, their approach verifies the correctness of semantic Web services composition based on models of Colored Petri Nets that are transformed from OWL-S models. It is sound to use such Petri Nets in order to verify reachability and soundness of composed services (among others). This approach differs from ours as it does not address our main topic — V&V of service composition and business processes.

In the dynamic field of late binding and runtime verification of business processes, Angelis et al. [2] introduced automatic test-case generation aiming at checks of the behavior of services participating in a given orchestration. Assuming that the business process is available as a runnable model, their approach applies model-checking techniques to derive test cases suitable to detect possible integration flaws. Our work does not rely on tests at runtime to find flaws, but uses formal logic at design-time for automated verification.

In the context of verification of business process models, Weber et al. [24] addressed the problem that control flow does not capture what the process activities actually do when they are executed. So, they annotated individual activities with logical preconditions and effects, specified relative to an ontology with axioms of the underlying business domain. This allowed them to verify the overall process behavior, but without making business rules explicit as such and dealing with them separately from the activities. In contrast, Deutsch et al. [7] studied automatic verification of data-centric business process specifications. Their results suggest that significant classes of data-centric business process specifications may be amenable to automatic verification.

The concept of business rules is extensive and to provide a clear, definitive definition proves to be difficult. According to Huang [10], business rules can be seen as operational rules that describe how an organization performs miscellaneous tasks. A similar specification has been given by Ross [18], where business rules are defined as the basic knowledge of a business including terms, facts and rules. The Business Rule Group defines business rules in the report [9] as statements that define or constrain the business of an organization. Their purpose is to influence the business process in a certain way.

However, business rules do not necessarily need to specify how an organization performs tasks but can also describe technical aspects. Orriens et al. [16] describe in their approach how business rules can be categorized and how such rules can be used for Web Service orchestration. They show that it is not sufficient to embed business knowledge in an orchestration language (like BPEL) directly, but that business rules need to be specified explicitly. This issue has also been discussed by Rosenberg et al. [17], who show the need for integrating business rules into BPEL, since they are often changed and thus a rule-based system should be used. Eijndhoven et al. [23] provide a similar motivation with the focus of business rules on action rules. Wu et al. [25] propose a rule-based scheduling engine that can be embedded into frameworks. The idea is to use rules as role assignments, which are used to influence process execution, message exchange, and flow constraints. None of these approaches deals with formal verification based on logic, however, like our work.

Lovrencic et al. [20] describe business rules as essential parts in today’s business system model and the need for their formalization. Two approaches based on UML and ontology-based modeling are discussed in this work.

In summary, we are not aware of any previous work that studied semantic service specification for V&V of service composition and business processes as we present it in this paper, especially not by including business rules with semantic specifications into formal verification.

4 Specifying Semantic Knowledge for Service Composition

In our running example, let us assume a very small (hypothetical) company, where in a given Business Case an Invoice is simply created first and then sent out. The services involved for that are specified above (somewhat simplifying, of course) in Listings 1 and 5. The formulations in Fluent Calculus are shown in Listings 2, 3, 4, and 6, respectively. The obvious service composition resulting in a sequential business process is shown above in Figure 1. A verification of this simple business process corresponding to a service composition was done against the specification of the atomic services using FLUX as shown above, and it succeeded. So far, everything looks fine, and the semantic knowledge for straight-forward service composition appears to be the same as for a business process.

Now let us assume a larger (hypothetical) company, where an Invoice needs authorization. For modeling the services, both CreateInvoice and SendInvoice can be reused, but an additional service needs to be specified. The additional service is presented in Listing 8 and specifies the authorization of a given Invoice.
AuthorizeInvoice:
Input: Invoice
Output: Invoice
Precondition: none
Postcondition: authorized(Invoice)

Listing 8: AuthorizeInvoice Specification

The formulation in the Fluent Calculus is shown in Listing 9.

AuthorizeInvoice:
poss(authorizeInvoiceOperation(Invoice), Z) :-
knows_val([Invoice], invoice(Invoice), Z).
% Input %

state_update(Z1, authorizeInvoiceOperation(Invoice), Z2, []) :-
update(Z1, [invoice(Invoice), isAuthorized(invoice(Invoice))], [], Z2).
% invoice(AuthorizeInvoiceOperationOutput)
is the output of the process/action and
isAuthorized(AuthorizeInvoiceOperationOutput, true) is the postcondition %

Listing 9: AuthorizeInvoice encoded in FLUX

The obvious business process for this larger company integrates the AuthorizeInvoice process between CreateInvoice and SendInvoice as seen in Figure 2.

Figure 2: Large Company: “Create, Authorize and Send” Business Process

A verification of this business process corresponding to a service composition was done against the specification of the atomic services using the FLUX tool, and it succeeded.

In principle, other business processes may be erroneously defined (which is more plausible and likely for large and complex processes, of course), such as the invalid processes shown in Figures 3 and 4. Somewhat surprisingly, also the verification of these service compositions for (invalid) business processes succeeded, although any reasonable validation would fail for them, of course. Note, that such a validation is not tool-supported and would in practice require expert business knowledge, while for our intentionally simple example it can be done using common sense. The reason for the process in Figure 3 is lack of explicitly represented knowledge, i.e., authorization is required before sending. The reason for the process in Figure 4 is also lack of explicitly represented knowledge, in this case it does not make sense in reality to authorize an already sent Invoice.

SendInvoice:
Input: Invoice
Output: none
Precondition: authorized(Invoice)
Postcondition: sent(Invoice)

Listing 10: SendInvoice Specification with explicit precondition

AuthorizeInvoice:
Input: Invoice
Output: Invoice
Precondition: sent(Invoice, false)
Postcondition: authorized(Invoice)

Listing 11: AuthorizeInvoice Specification with explicit precondition

These additional preconditions are represented in FLUX as well and the new specification encoded in Fluent Calculus is shown in Listings 12 and 13.

SendInvoice:
poss(sendInvoiceOperation(Invoice), Z) :-
knows_val([Invoice], invoice(Invoice), Z).
% Input %
\( \text{holds}( \text{isAuthorized}(\text{invoice}(\text{Invoice})), Z). \% \text{Precondition} \%
\)

\( \text{state\_update}(Z1, \text{sendInvoiceOperation}(\text{Invoice}), Z2, []) :-
\)
\( \text{update}(Z1, [\text{isSent}(\text{invoice}(\text{Invoice}))], [], Z2). \% \text{Postcondition} \%
\)

Listing 12: SendInvoice Specification with explicit precondition in FLUX

\( \text{AuthorizeInvoice} :\)
\( \text{poss}(\text{authorizeInvoiceOperation}(\text{Invoice}), Z) :-
\)
\( \text{knows\_val}([\text{Invoice}], \text{invoice}(\text{Invoice}), Z), \% \text{Input} \%
\)
\( \text{knows\_not}(\text{isSent}(\text{invoice}(\text{Invoice})), Z). \% \text{Precondition} \%
\)

\( \text{state\_update}(Z1, \text{authorizeInvoiceOperation}(\text{Invoice}), Z2, []) :-
\)
\( \text{update}(Z1, [\text{isReadyForSending(\text{Invoice})}], [], Z2). \)

Listing 13: AuthorizeInvoice Specification with explicit precondition in FLUX

Strictly speaking, however, another service composition for an (invalid) business process as shown in Figure 5 was still verified with FLUX, but certainly not validated.

5 Making Business Rules Explicit

Actually, this is business knowledge in addition to these services, more precisely these are a kind of business rules. So, we propose to make such knowledge explicit in an additional specification separate from the service specification.

Since the Fluent Calculus essentially works with operations, we model such a business rule as an operation that sets a specific state, which is its postcondition, according to its input and precondition. This does not entail that some specific business actor would have to perform such an operation. It just models the missing business knowledge in such a way that it fits the given formalism required for automated verification.

Listing 15 shows an example of such a business rule. It sets for a specific invoice (its input) that it is ready for sending. The rule is only applied if an invoice is present and has already been authorized. Comparing this listing with the service specification in Listing 10 shows that the additional precondition has been moved to an explicit specification of our business rule.

\( \text{isReadyForSending\_BusinessRule} :\)
\( \text{poss}(\text{ruleIsReadyForSending}(\text{Invoice}), Z) :-
\)
\( \text{knows\_val}([\text{Invoice}], \text{invoice}(\text{Invoice}), Z), \% \text{Input} \%
\)
\( \text{holds}(\text{isAuthorized}(\text{invoice}(\text{Invoice})), Z). \)

\( \text{state\_update}(Z1, \text{ruleIsReadyForSending}(\text{Invoice}), Z2, []) :-
\)
\( \text{update}(Z1, [\text{isReadyForSending(\text{invoice}(\text{Invoice}))}], [], Z2). \)

Listing 15: Business rule that determines if an Invoice is ready for sending

However, this explicit specification of the business rule requires some minor changes to the service specifications. The idea is that every service sending an invoice has an additional precondition that checks if a specific invoice is, in fact, ready for sending. Only if this condition is fulfilled, the operation can be invoked. In our example, this condition is directly related to the postcondition of the business rule defined in Listing 15. Listing 16 shows the additional precondition isReadyForSending encoded in FLUX. The results of the verification stays the same. That is why the state\_update part has been omitted. This approach can be applied to all service specifications at once, while the service implementations do not have to be changed.

\( \text{SendInvoice} :\)
\( \text{poss}(\text{sendInvoiceOperation}(\text{Invoice}), Z) :-
\)
\( \text{knows\_val}([\text{Invoice}], \text{invoice}(\text{Invoice}), Z), \% \text{Input} \%
\)

In fact, this additional knowledge encoded is not directly related to these services per se.
Listing 16: SendInvoice plus precondition for business rule encoded in FLUX

However, there is a problem with the specifications of the previous listings. Process 6 also verifies with this specification. The problem is that the precondition of the sendInvoiceOperation still holds after its invocation and thus it can be invoked again. To solve this problem, the fact has to be explicitly removed from the knowledge base that the invoice is ready for sending. The resulting specification is shown in Listing 17, and with this specification Process 6 does not verify anymore.

Listing 17: Amended SendInvoice Specification encoded in FLUX

The same approach is also applied to the authorization operation defined in Listing 14. Here we can introduce a new business rule that sets for a specific invoice if it is ready for authorization. The specification of the operation must then include this fact as a precondition.

Listing 18 shows this additional business rule. In this case, the business rule is quite simple and just specifies that all invoices are automatically ready for the send operation. This rule is shown in Listing 20.

Listing 18: Business rule that determines if an Invoice is ready for authorization

Listing 19 shows the adjusted service specification. The additional precondition has been introduced and also the result of the operation has been adjusted, so that the fact that the invoice is ready for authorization is removed after the invocation of the operation.

Listing 19: Adjusted AuthorizeInvoice operation

With these explicitly specified business rules, only the valid Figure 2 can be verified, while the verification correctly fails for all others. However, there is still an issue with our small company that does not have an authorization operation in its business process. So, Figure 1 still cannot be verified. Since there is no authorization operation, the business rule has to be adjusted. Similarly to the business rule in Listing 18, we can specify a business rule that states that all invoices are automatically ready for the send operation. This rule is shown in Listing 20.

Listing 20: Business rule that determines if an Invoice is ready for sending in a small company

In effect, there is no need to change the service specification anymore. The specifications shown in Listings 16 and 17 can be directly reused.

6 Discussion and Future Work

Business rules can be implemented in WS-BPEL (Business Process Execution Language), a popular orchestration language for Web services, which includes procedural
constructs such as loops and conditional selection of services [15]. However, in contrast to our approach, business rules are not identifiable as such in the procedural code. So, it is very difficult to extract what a business rule exactly states, and they are difficult to reuse. This approach is inherently different from ours as presented above and requires a different form of verification.

Business rules can also be implemented within services. However, this requires knowledge about variability for different processes in their specification (possibly with conditional preconditions) and of all tacit business rules as well. And it reduces the reusability of such services and contradicts the intention of the service approach. We showed above that simply adding knowledge of a business process to a semantic service specification makes it specific to this process and does not allow the reuse of this service in another business process any more.

Therefore, we strongly argue for the separation of service and business rule specification, which is a major contribution of our paper. This does not contradict, however, the uniform representation of services and of this kind of business rules in Fluent Calculus, so that the FLUX tool was able to use them together for formal verification.

Based on this V&V approach, even another approach to service design may be taken. Instead of directly providing the usual WSDL specification and a procedural implementation of a service, a semantic specification may serve as a starting point. Such specifications can be immediately checked based on our V&V approach for services and business processes. Only after they are considered appropriate, corresponding service implementations should be provided. Since formal specifications of pre- and postconditions are available, even formal correctness proofs of the procedural code of the service implementation may be performed based on them. These would start from the postcondition of a service and even automatically determine the weakest precondition for the given code, which can be checked whether it matches the given precondition. Still, loop invariants may have to be provided additionally for such a proof.

While this paper deals with verification of service composition, it is also possible to provide an automatic composition of services based on our combination of semantic Web-service specifications and business rules. Since FLUX also provides the possibility to automatically generate plans and even to develop an additional planner, it is possible to automatically generate service compositions for achieving a given goal condition. These would be verified by definition through this way of being generated. In addition, when a complete set of additional business rules is employed, the related business processes would be valid, too. Of course, the completeness of business rules cannot really be guaranteed, so that a validation of generated service compositions as business processes will still be required.

It is also possible for FLUX to have multiple operations to be triggered by a single operation. In such a situation, it creates all possible plans, from which a valid one in the sense of the business process may be chosen. FLUX can even deal with concurrency. For further information on the Fluent Calculus implemented by FLUX, we refer the reader to the FLUX manual at http://www.fluxagent.org. Overall, our verification approach using FLUX can handle everything that FLUX can.

This planning feature of FLUX can also be used for a different kind of verification approach. In this paper, we focus on the verification of given sequences of services, but FLUX also provides the means for a verification of whether a goal condition can be achieved by such a sequence.

As mentioned in the Related Work section above, OWL-S can be used as a starting point for a translation of OWL-S service definitions to FLUX. These service definitions can then be used with our approach for an automatic verification using Fluent Calculus. However, to our best knowledge, the inclusion of business rules in such an OWL-based approach has not been studied yet. According to our own preliminary study, business rules have to be specified in OWL or a related language. Analogously to the approach presented in this paper, certain business rules may be represented in OWL-S like services. However, this approach might conflict with the philosophy of OWL-S, where services are the main focal point. Alternatively, since SWRL is often used in combination with OWL, business rules may also be represented in this rule markup language. Business rules specified with SWRL can then be used to generate the corresponding FLUX plans during the translation of the OWL-S services. The idea is that the SWRL rules provide in their body part some preconditions that have to be fulfilled before the service in the head part can be invoked. During the translation to FLUX, these preconditions in the SWRL rules are then encoded as additional preconditions for the service that is defined in the corresponding head part of the SWRL rule.

The extensive use of logic in this paper is just a means to an ends, formal verification. In fact, it would be more difficult without a formal logic and a related tool to uncover the tacit knowledge required for verification that matches validation. The issue of missing business knowledge that would over-specify service specifications is more fundamental and inherent for service composition in the context of business processes.

7 Conclusion

In this paper, we show with simple examples that tacit business knowledge needs to be made explicit and even for-

Semantic Web Rule Language (SWRL): http://www.w3.org/Submission/SWRL/
mally represented for automated verification and validation (V&V) of service composition and business processes. This otherwise missing knowledge is a kind of business rules that only partly will be ever thought about in practice. Traditionally, the only place to represent such missing knowledge explicitly would be in the semantic service specifications. Even our simple examples reveal, however, that this may lead to mismatches between these specifications and the service implementations in the sense of over-specifications. So, this would be the case in any reasonable real-world process as well. This means that semantic service specification is not sufficient for V&V of service composition and business processes, since (at least) certain business rules need to be specified additionally and separately.

References


