The Journal of Systems and Software 137 (2018) 217–238

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Compositional execution semantics for business process verification

Emmanouela Stachtiari*, Panagiotis Katsaros

Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 20 January 2017
Revised 1 November 2017
Accepted 4 November 2017
Available online 2 December 2017

Keywords:
Formal verification
Programming language semantics
WS-BPEL
BIP

ABSTRACT

Service compositions are programmed as executable business processes in languages like WS-BPEL (or BPEL in short). In such programs, activities are nested within concurrency, isolation, compensation and event handling constructs that cause an overwhelming number of execution paths. Program correctness has to be verified based on a formal definition of the language semantics. For BPEL , previous works have proposed execution semantics in formal languages amenable to model checking. Most of the times the service composition structure is not preserved in the formal model, which impedes tracing the verification findings in the original program. Here, we propose a compositional semantics and a structure-preserving translator of BPEL programs onto the BIP component framework. In addition, we verify essential correctness properties that affect process responsiveness, and the compliance with partner services. The scalability of the proposed translation and analysis is demonstrated on BPEL programs of various sizes. Our compositional translation approach can be also applied to other executable languages with nesting syntax.

1. Introduction

Businesses rely more and more on distributed, value-adding software applications in order to offer enterprise functionality to customers. Business Process Modeling (BPM) is a promising paradigm for integrating software components into a single executable unit, termed as process. The Service-Oriented Architecture (SOA) suits to the BPM paradigm, with respect to the composition of services into processes, which can be also deployed as services. Among existing languages for the specification of such processes, BPEL stands out by providing high-level primitives, and constructs for the definition of complex synchronous and asynchronous web service interactions. The used web services are autonomous and loosely-coupled components that possibly span different organizations. For the wide adoption of business process programming, it is essential to ensure reliability in order to avoid errors that may cause critical losses to the involved organizations. Additionally, the program has to fulfill correctness goals such as process responsiveness and compliance with partner services.

One approach towards ensuring reliability is by testing the process with emulating its interactions (Sun et al., 2015). In this case, an adequate coverage of the program's control flow has to be achieved by selecting the appropriate test inputs. On the other hand, formal verification guarantees full coverage of execution paths for all possible inputs. Such an analysis has to be based on a formal specification of the language execution semantics, which involves nesting of service interactions using concurrency, isolation, compensation and event handling constructs.

Many works attempt to verify correctness by model checking a *formal model*, which is an abstract representation of the service composition program (Beek et al., 2007). However, the original structure of the source program is not reflected in the formal model, thus rendering impossible to exactly locate the verification findings in the program's code. This is an inherent problem of most formalisms, which lack sufficiently expressive composition primitives for a model representation that preserves the service composition structure. The BIP (Behavior, Interaction, Priority) component framework (Basu et al., 2011b) provides a minimal set of primitives adequate for preserving the service composition structure. It consists of an executable modeling language for layered transition systems, which has formally defined operational semantics and mathematically proven expressiveness (Bliudze and Sifakis, 2008). The BIP models can be formally verified with the BIP tools (BIP tools, 2017).

We use BIP to introduce a compositional semantics for BPEL , i.e. a semantics in which the processing for each BPEL construct is placed locally to a corresponding BIP component. Such a definition tackles the combinatorial problem of defining semantics for each possible combination of nested BPEL constructs. Compositional semantics can be defined for executable languages with nesting syn-

* Corresponding author.
E-mail addresses: emmastac@csd.auth.gr (E. Stachtiari), katsaros@csd.auth.gr (P. Katsaros).

https://doi.org/10.1016/j.jss.2017.11.003
0164-1212/

tax if the execution semantics of enclosing and nested constructs can be defined independently from each other. To achieve such a definition in our approach, the semantics of nesting constructs are defined based on abstractions built-in by construction for the nested ones, while the latter are combined using coordination primitives that do not alter their semantics (just restrict their execution traces). A structure-preserving translator into the BIP language has been implemented that covers all activities of the BPEL standard. The translator transforms the BPEL programs into BIP models that contain the code needed for the verification of essential correctness properties. The check of whether the properties are met takes place by exploration of the reachable state space. If a property is violated, we are able to obtain a counterexample execution trace that contains the processing steps of BPEL activities, which lead to the error location.

In Stachtiari et al. (2012), we presented a first version of our translator for a limited set of BPEL constructs with more emphasis on the translation algorithm. The verification of a functional property for a showcase application scenario was also demonstrated along with evidence for its violation in the form of a counterexample. Here, we expose:

- the complete execution semantics of BPEL through a new methodology for compositional definition;
- the verification of a wide range of important correctness properties;
- the testing of our translator in mid-scale programs and their verification.

We note that the translation times were found to have a statistically significant *linear relation* to the number of states of the generated BIP model. The translator, the verification utilities for the properties of interest, as well as the BPEL programs of our experiments are available online in BPEL2BIP (2017). Verification is only one of the possible uses of our BPEL process models, which can be also used e.g. for test case generation based on the produced execution paths (Jehan et al., 2015). Moreover, in an independent research work (Ben Said et al., 2016), our approach was extended towards enabling the configuration of information flow policies for BPEL processes. Finally, our BPEL process models can run as standalone web services on top of the BIP engine, after being enhanced with runtime communication support (e.g. connections, dispatching) based on the architecture for SOAP-based web services that we proposed in Stachtiari et al. (2014).

In Section 2, we discuss the design problems and the correctness of BPEL processes through a motivating example. Section 3 introduces the structure of our BIP model and the principles of the compositional approach for the definition of the BPEL execution semantics. These principles determine the interface and the behavior of BIP components, which allow implementing the semantics of the various BPEL activities. Section 4 encodes the BPEL execution semantics into safety properties that are *enforced in our model by construction*. Our modeling approach covers all activities of the BPEL standard, but the presentation is restricted to the most important activities and details for more activities are exposed in Appendix B. In Section 5, we present the verification of essential correctness properties that have been previously introduced in Section 2 and the formalization of additional useful correctness properties. Section 6 discusses the principles of the translation of BPEL programs in BIP. Section 7 shows results from the translation and analysis of mid-scale BPEL applications and the paper concludes with a critical review of the related work in Section 8 and our remarks for the exposed contributions in Section 9.

2. Correctness of BPEL processes: a motivating example

BPEL process implementations are based on web services (partner links) whose interfaces expose *service operations* written in the WSDL 1.1 language. Synchronous operations accept an input and block the invoker for the output, or a fault, to be returned. On the contrary, in asynchronous operations the invoker dispatches the input and forgets it. Thus, through the use of two asynchronous operations it is possible to apply a request-response interaction pattern that does not block the invoker. In this approach, a service is invoked with the first operation and the response is returned with a second operation, referred to as *callback*, exposed by the invoker. The use of asynchronous operations generally allows for complex service interaction patterns, such as parallel operation invocations, but it raises the need to effectively manage communication *sessions*, i.e. the stateful chains of dual service interactions. The assignment of messages to the correct session takes place by message *correlation*.

Atomic behavior in processes is realized with *basic activities*, such as the `invoke`, `receive`, and `reply`, which are used respectively to (i) invoke, (ii) receive input, and (iii) send output (or fault), with respect to specific service operations. Fig. 1a and b show the client-side and server-side activities used for a synchronous (resp. an asynchronous) invocation of an operation `x`. A client-side synchronous invocation is implemented by a request-response `invoke`, while the asynchronous interaction relies on an one-way `invoke` of `x` and a `receive` of the callback operation `y`. Generally, the `assign` activity is used before sending and after receiving a message, in order to copy data between the message and the process's variables. BPEL 's *structured activities* define workflows of activities, such as `sequence`, parallel `flow`, and other conditional and repeatable structures. The `scope` activity defines a local context for its enclosed activities, with its own data and error handling through compensation, termination and fault handlers. A scope also defines event handlers for incoming messages and timeouts.

Example 1. A BPEL process for travel booking is presented in Fig. 2 with its activities shown in rectangular boxes. The activities for service interactions are labeled with the invoked operations. The bold, the thin and the dotted edges represent respectively relationships for the order of execution, the containment of handlers and the synchronization between activities.

The process provides to its clients the synchronous operation *get_itinerary* that responds with an output or a fault message. When a client wants to book a travel itinerary, a *get_itinerary* request is received along with the preferred hotel, room type and flight details. Two scopes are then executed in parallel that communicate respectively with the *HotelBookWS* and *AirlineBookWS* web services:

- The *Hotel-booking* scope invokes the asynchronous *bookHotel* operation of *HotelBookWS* to reserve the chosen hotel room. For this purpose, it uses an one-way `invoke` and continues its processing, while the response is pending. A `receive` waits for the confirmation in the *hotelBooked* callback operation. When the confirmation is received, the synchronous *payHotel* operation of the *HotelBookWS* is invoked for the payment. The progress of the whole process is then blocked on the synchronous `invoke`, until the receipt of the expected response. In parallel to the normal flow, the scope also has an event handler that listens to requests for the *noAvail* operation. This is a callback operation that is invoked by the *HotelBookWS* service, if there is no availability for the chosen hotel room. Upon receipt of such a message, the event handler throws a *bookFailed* fault.