Hybrid analysis of BPEL models with grammars

Erwin de Jager[1] and Stijn de Gouw[2]

[1] Belastingdienst, The Netherlands
[2] Open University, The Netherlands

Abstract. BPEL (Business Process Execution Language) is a language to formally describe business processes, and synthesize suitable executable orchestration code for web-services accordingly. In particular, the subset of BPEL of so-called straight-through processes (STP), tailored for short-living processes, is used in banks and tax administrations to process large numbers of transactions per hour without human interaction. This paper contributes two novel tool-supported approaches for the analysis of STP event traces. Both are based on (attribute) grammars: a static verification approach based on parsing and a Prolog-based approach for automatic test-case generation. Our tool suite supports both protocol and data-oriented properties of STP event traces. We validate and compare our tool suite to existing approaches with an industrial case study of the Dutch Tax and Customs Administration (Belastingdienst).

1 Introduction

BPEL is a widely used standard in organisations such as banks or tax administrations to formalize business processes, and synthesize executable code accordingly. BPEL processes run in a service-oriented environment, interacting with the environment exclusively through web service interfaces. This paper contributes two novel tool-supported analysis approaches to verify both protocol and data-oriented properties of short-running BPEL processes (also called STP or Straight-Through Processes). Protocols specify valid BPEL event orderings (but they may depend on data!), and data-oriented properties can be used to specify for example parameter and return values of invoked web-services. Both analyses are based on the declarative formalism provided by (attribute) grammars: a static verification approach based on parsing, and a Prolog-based program for automatic test-case generation.

Related work BPEL verification has been the subject of numerous investigations. In Section 4 we carry out an extensive comparison with existing tool-supported approaches. Here, we briefly describe other research. Morimoto [9] states that

due to the high level of abstraction of BPEL models, accuracy cannot be guaranteed and formal models are required for verification. The use of languages such as BPEL provides a complete specification for service orchestration, but does not enforce correctness. This article compares verification of BPEL through three formal models: PETRI, ALGEBRA and AUTOMATA. Formal models can reduce complexity of the composition and provide a mathematical basis that could be used to prove certain properties. Other research focuses on simulation of the BPEL run-time environment [3]. This allows test cases to be verified in a controlled manner. This approach provides for the testing of run-time aspects such as system performance and behaviour in the case of web services that do not respond correctly . Another approach is to verify BPEL from the logic or an axiomatic system [8, 1].
Kokash et al. [7] present a data aware verification approach for Reo, a coordination language for which converters from BPEL are available. They do not discuss to what extent the conversion from BPEL to Reo supports data, in particular how to handle protocols whose communication behavior depends on data. In our paper this is supported through so called semantic predicates. Based on 177 articles, Bozkurt [2] describes the pros and cons of test strategies for SOA and associated tools. Bokurt states that Service-centric system testing is more challenging compared with traditional systems for two primary reasons: the complex nature of Web services and the limitations that occur because of the nature of SOA. In particular, the following issues are identified that limit the testability of service-centric systems: limitations in observability of service code, lack of control due to independent infrastructure on which services run, dynamics and adaptiveness that limit the ability of the tester to determine the web services that are invoked during the execution of a workflow, and cost of testing. Our research aims to alleviate all but the second of these issues.

2 BPEL models and grammars

Organisations such as Banks and Tax administrations process large numbers of transactions per hour without human interaction. One of the main languages used to optimize and formalize such processes in an executable model is BPEL (Business Process Execution Language), as described in the WS-BPEL 2.0 standard [12]. BPEL is an XML based language with the goal to implement a business process by orchestrating Web Services. A second objective is the efficient production of automated business processes through the use of formal models at the abstract level of end-users and the reuse of Web Services. The application of BPEL is one of the possibilities for achieving a service oriented architecture (SOA). A business process defined in the BPEL language can be executed by a BPEL engine. In most cases, a stimulus in the form of a message initiates the start of the business process. A number of process steps will then be carried out until the process reaches an end state. Two kinds of of processes are distinguished: long-running and short-running processes. A long-running process can be active for many days and often includes process steps with asynchronous links

to users or other systems. The long-running service life requires the persistence of the status and other attributes of the process. Short-running processes have a very short life span and only have active and inactive states. This type of process is intended for high-volume transaction processing, referred to as *STP*. Short-term processes therefore use synchronous links. Only in the initial and final state of a process asynchronous connections can be used. STP processes are lightweight and capable to process large numbers of messages in parallel, achieving the required throughput. Those cases that are not provided for, or fail, terminate as an exception. These cases end up in an office process where data enrichment takes place. The case is then offered again to the STP process. From a functional point of view, an STP process is simple: all implementation details are delegated to the called web services.

As an example, Figure 1 shows (in a graphical notation) a BPEL process used in the dutch tax administration for handling requests, for example in the form of an e-mail or a letter. The process starts after an request is received. The arrows depict the direction of the execution flow and diamonds represent choices. This request is processed step-by-step until one of the end states ("Request handled" or "Request rejected") is reached. The shown process includes exception handling and re-injection of failed cases: after the (cause of the) failure is resolved, a previously failed case can be re-injected in the process. Invocation actions refer to services and data definitions defined in external WSDL files (an XML-based languages to describe the signatures of the operations a web service provides). The example clearly shows the scope of BPEL, that is, orchestration of web services.

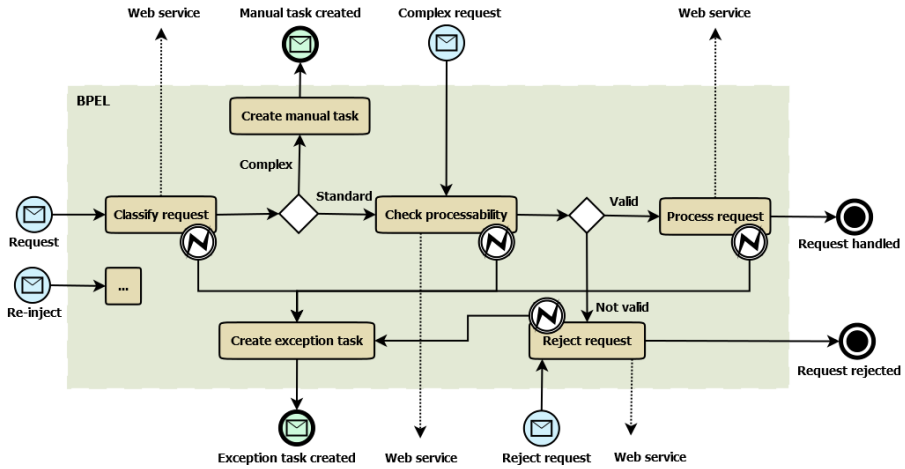

Fig. 1. BPEL Request Handling STP process

2.1 Grammar-based Verification

To analyze BPEL STP processes, we transform a given BPEL to a (attribute) grammar. In the simplest form, the workflow within a BPEL model is transformed to a context-free grammar in which the BPEL events are represented by grammar terminals, and the non-terminals of the grammar capture the allowed orders of events. From the grammar, a parser is generated that checks whether a trace of BPEL events matches the grammar. Figure 2 summarizes this approach.

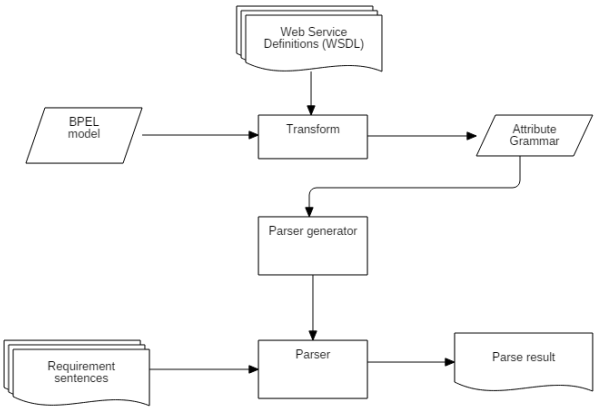

Fig. 2. Grammar based BPEL verification

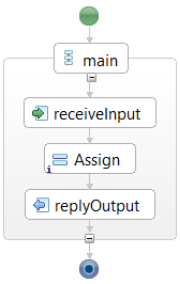

Fig. 3. Minimal case

Henceforth, we call BPEL event traces *requirement sentences*. As a minimal example of the transformation, the BPEL process in Figure 3 is transformed into the grammar with a single production: $S ::=$ receiveInput assign replyOutput.

Table 1 shows the main BPEL constructs and their corresponding translation to an attribute grammar in the syntax of the popular ANTLR parser generator. Attribute grammars are an extension of context-free grammars by Knuth [6] with data. The translation proceeds recursively, transforming the different BPEL XML elements into grammar constructs. The full translation can be found in an extended version of this paper on Github [4].

Verifying protocol properties by parsing requirement sentences is useful but incomplete without the data that control the process. By enriching the grammar with attributes, the correctness of the STP process can be assessed while also taking data and data transformations into account. In particular, in the grammar, productions that depend on a condition on the data (as captured in the grammar attributes) are represented by semantic predicates [10] in the grammar productions. They have the form $\{b?\}$, where b is a boolean expression on the attributes, and the production in which the semantic predicate appears is only applicable if b evaluates to true.

BPEL Element	**Grammar**	**Additional processing**
<bpel:process name="X" ...	grammar X;	
	x : v0 < children > v;	x = X.toLowerCase(), transform children
<bpel:pick name="X"	'X' (< $onMessage_1$ > | ... | < $onMessage_n$ >)	transform onMessage children
<bpel:onMessage operation="X"	< children >	transform children
<bpel:if name="X"	'X' ({ < condition >? } < children >)	use this rule if followed by else or elseif and transform children
	'X' (< children > | { not < condition >? })	use this rule if not followed by else or elseif, transform children, get < condition > from context and add semantic predicate
<bpel:else	| ({ not < condition >? } < children >)	get < condition > from context, add semantic predicate, transform children
<bpel:elseif	| ({ < condition >? } < children >)	transform children
<bpel:condition>	{ < condition >? }	transform XPATH condition to Java, add condition to context
<bpel:copy>	< to > = < from >;	transform to and from children
<bpel:to>	< Java expr >	transform to element to Java expr
<bpel:from>	< Java expr >	transform from element to Java expr

Table 1. Transform BPEL protocol to a grammar

The data transformations can be derived from the BPEL model. The results of the called web services can not be derived as they depend on parameters and the internal functionality of the web service. The results could be serialized from production logs, or simulated by mocking the service, or specified using assertions. The Web Service Definition Language (WSDL) file of a web service contains a formal description of the interface. The transformation step uses this interface to capture the data involved in the BPEL process as attributes in the grammar, and the requirement sentences are enriched with input and data to emulate web service calls.

WSDL Element	**ANTLR Element**	**Side condition**
<element name="X"> <complexType name="X">	class X {<children>}	
<element name="X" type="T"/> <complexType name="X">	T X; T X = new T(); T getX(){}, setX(T x){}	T is Java primitive type T is Java complex type create getter and setter
<extension base="X">	extends X {<children>}	extends with baseclass X
<part element="X" name="Y">	X y = new X(); X getY(){}	'y' is used in BPEL model
<message name="X">	class X_ {<children>} X_ x = new X_();	X_ unique name 'x' is used in BPEL model
<sequence>..</sequence> <complexType>..</complexType> <types>..</types> <schema>..</schema> <complexContent>..</complexContent>	<children>	

Table 2. Transform BPEL data elements to a grammar

The final step in the grammar-based analysis consists of verifying properties of the data in the BPEL process. Suppose that $\mathsf{V} = \{\mathsf{v}_0, \mathsf{v}_1, \mathsf{v}_2, ..., \mathsf{v}_n\}$ is the set of all variables within an STP process and that PS_i represents a process step i. Then $\mathsf{PS}_i(\mathsf{V}, \mathsf{V}')$ defines the data transformation within process step i (those are translated in e.g. the copy/to/from elements and yield updated values of the attributes). V and V' contain respectively the initial values and the final values of the variables after execution of step i. By adding $\mathsf{V}_{\mathsf{initial}}$ and $\mathsf{V}_{\mathsf{expected}}$ as attributes to the grammar, these variables can be used as preconditions and postconditions. After parsing completes and all variables have been updated during parsing (possibly repeatedly), the parser checks in the last step whether the condition ($\mathsf{V}_{\mathsf{actual}} == \mathsf{V}_{\mathsf{expected}}$) holds.

2.2 Test-case generation

We have developed a method to generate test cases using the logical programming language Prolog in combination with so-called Definite clause grammars

WSDL: https://www.w3.org/TR/2001/NOTE-wsdl-20010315

(DCG) [11]. In a DCG, each production rule has the form: $nt \rightarrow body.$, where nt is a non-terminal symbol and body is a sequence of one or more terminals/non-terminals separated by commas. A non-terminal symbol is written as a Prolog atom, while a sequence of terminals is written as a Prolog list, where a terminal may be any Prolog term. By defining the generated grammar as a DCG within Prolog, test cases can be derived that meet syntactic requirements of the ANTLR grammar. The ANTLR grammar also uses data transformations and semantic predicates to enforce valid paths. DCGs do not have these features, they generate syntactically correct paths without taking conditions and error situations into account. Therefore, some of the paths generated will not be accepted by the parser. DCGs are supporting in this way a test approach called Fuzzing as described by Holler [5]. This approach uses variations on correct input to detect errors in software.

Initially all variables in both the pre and post conditions are under-specified and set to the default value '?'. These variables will be added to the paths generated by BPEL2DCG. All test cases will be rejected by the parser. Rejection may be for three reasons. In the first place because paths are not valid. If the expert endorses the parser's verdict then these non-valid paths can be removed. Secondly, when post conditions variables do not contain the expected values. If the expert judges the value determined by the parser to be correct, then the value must be applied as a post condition. A test case can also fail because it does not meet a semantic predicate of the parser. In that case, the expert may decide to adjust the corresponding pre-conditions. Adjusted test cases must be re-verified by the parser. These steps must be repeated until all test cases are accepted by the parser.
If the expert observes that the parser approves a test case incorrectly, then the BPEL model is under-specified. It can also be observed that the parser incorrectly rejects a test case. In that case the BPEL model deviates from the requirements. After performing this procedure, these created test cases should fully cover all requirements and can be used for regression testing.

3 Implementation

We implemented the grammar-based verification and test-case generation described in Section 2 in our own tool suite. The architecture of the grammar and test-base verification tooling is shown in Figure 4.

All steps in Figure 4 can be performed automatically except for activity 'Modify and remove testcases'. In that step, a domain expert interprets the results of running a set of test cases, for example as a regression test (which test pass/fail, and is this expected?) and can potentially modify the test suite based on their interpretation of the results. Clearly, this requires domain knowledge from the expert. See the paragraph on BPEL2DCG for more information. We next discuss the main components of our tool suite.

BPEL2CFG This component takes a BPEL model as input and generates an ANTLR grammar as output, following the transformation scheme given earlier

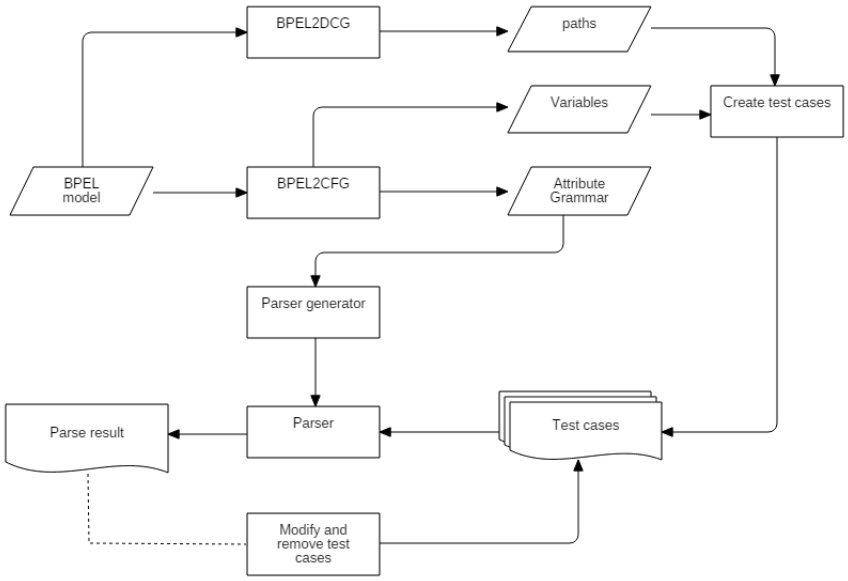

Fig. 4. Test case construction

in table 1 and table 2. It also provides a list of variables and values used in BPEL conditions. These conditions are applied within the ANTLR grammar as semantic predicates. It is implemented as a 900 line program in Eclipse Epsilon, a Model-to-model transformation tool that offers capabilities to transform an XML file that meets an XSD (i.e. the BPEL standard) to a target model (in our case: an ANTLR grammar). Because BPEL is specified in structured XML and can be interpreted directly by Epsilon. The flow of a BPEL model can be transformed in a fairly straightforward manner to productions of a CFG grammar.

Although the number of elements involved in data transformations is limited, the complexity is significantly higher. First, the variables defined in the BPEL model must be converted into Java implementations. Next, the BPEL operations on these variables have to be translated into a Java representation. Thus, the translation includes XPATH expressions, XML conventions and nested XML structures. Approximately 70% of all Epsilon lines or code of BPEL2CFG relate to data transformations.

BPEL2DCG This component transforms a given BPEL model into a Definite clause grammar G for use with Prolog.
The Prolog query **findall(X, phrase(G,X,[]),AS)** then generates all (finitely many, for STP processes) testcases that comply with the grammar. BPEL2DCG is implemented in Eclipse Epsilon and consists of 669 lines of code.

Eclipse Epsilon, https://www.eclipse.org/epsilon/

Parser generator The parser generator transforms a given grammar into a parser. We use the parser generator to check whether a trace of BPEL events conform to the specification as given in a grammar. We use ANTLR [10] as the parser generator. It is a popular and powerful parser generator which supports attributes (i.e. the data in BPEL processes) and semantic predicates for productions that are conditional on data values.

4 Case study

To assess our approach, we performed an industrial case study of the Tax and Customs Administration of the Netherlands (In Dutch: Belastingdienst) and compared with existing approaches. At the Belastingdienst, BPEL is used in several domains, including:

- Digitization of notary information: processing 1.6 million notarial acts yearly
- Motor vehicle tax: process 440.000 new license plate registrations, and 1.8 million mutations yearly
- Value added tax (VAT): worth € 50 billion yearly
- Payroll tax allowances: pay out € 850 million yearly to 120.000 employers

We verified formalized and analyzed several BPEL processes from the Belastingdienst (in multiple tools), but for space reasons we focus here on the most extensive case, named 'Handle Case Request' based on Payroll tax allowances and shown in Figure 5.

The vast amount of research done in verification of web services has resulted in many different approaches, some of which are supported by tooling [2, 9, 13]. Most common used formal models are Petri nets, Process Algebras and Automata. These three formalisms will be used to make a comparison with the grammar-based approach. The other tools first translate the BPEL model into their own formalism (i.e. Petri nets) and next verify the result with an appropriate model checker. Our grammar based approach does not use model checking, but generates a parser to verify the BPEL model.

Tool	**Purpose**	**Input**	**Output**	**Year**
BPEL2PNML	Translate BPEL to Petri nets	BPEL 2.0	Petri Net Markup Language (PNML)	2005
LTSA	A Tool for Model-Based Verification of Web Service Compositions and Choreography	BPEL 2.0	Finite State Process (FSP) process algebra	2006
BPEL2OWFN	Translate BPEL to Petri nets	BPEL 2.0	open Workflow Net (oWFN) (Petri Net)	2007
BPEL2STS	Symbolic Web Service Composition Testing	BPEL 2.0	Symbolic Transition System (Automata)	2011
BPELVT	A Tool for Formal Validation of Web Service Orchestrations	BPEL 2.0	Promela (Automata)	2012

Table 3. BPEL transformation tooling

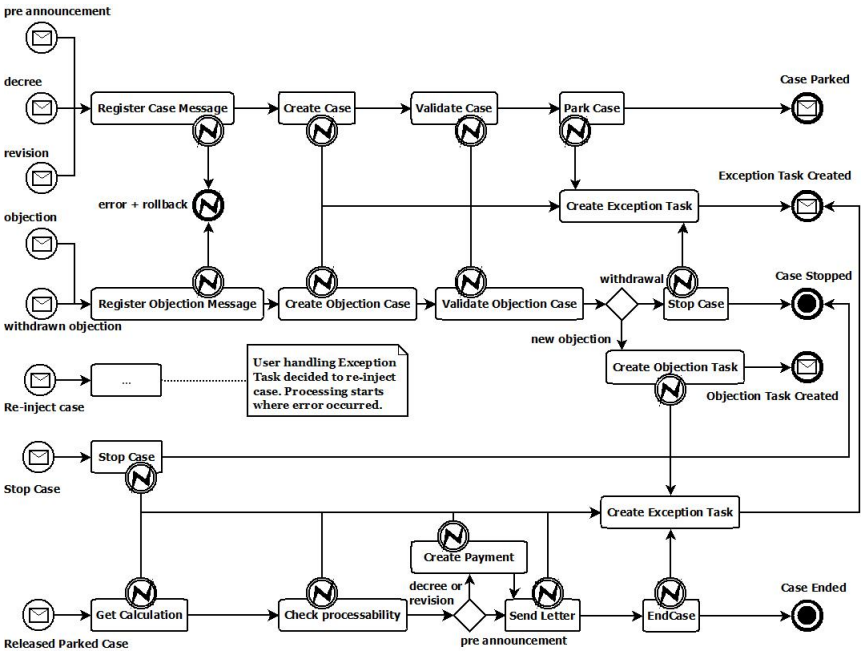

Fig. 5. BPEL process Case handling

Table 3 lists the existing tools we evaluated. These tools transform a BPEL model as input into a formal model, and verify this model with a model checker. We investigated whether (and to what extent) the tools supported the following categories of properties:

- **Correctness**: the path followed within a process is consistent with the input and the data returned by the called web services
- **Complete BPEL semantics**: does the tool cover the entire WS-BPEL specification?
- **Reachability**: Are all states of the STP process reachable?
- **Deadlock**: verify absence of deadlock
- **Liveness**: STP process eventually always reach an end state
- **Timing failures**: are timeouts of STP process or invoked services handled?
- **Traceable Model**: Each structural element in the formal verification model corresponds to an element in the original STP process

Results We found that applying the tooling developed in related work (table 3) on our case study was not possible without adjustments. The cases were developed with Eclipse BPEL Designer 1.1.1 (2017), which is more recent than the transformation tooling used. Certain XPATH features are not recognized by the tooling. This means that in most cases the generated BPEL code cannot be interpreted by the transformation tooling. However, after minor adjustments to the BPEL code, interpretation was possible.

Property	**Grammar**	**Petri Net**	**Process Algebra**	**Automata**
Correctness	bpel2cfg	-	-	-
Complete BPEL semantics	-	bpel2owfn LoLA	-	-
Reachability	(bpel2cfg bpel2dcg)	bpel2pnml wofbpel bpel2owfn LoLA	LTSA	-
Deadlock	-	bpel2owfn LoLA	LTSA	-
Liveness	(bpel2cfg bpel2dcg)	bpel2owfn LoLA	LTSA	-
Timing failures	-	bpel2owfn LoLA	-	-
Traceable Model	bpel2cfg bpel2dcg	-	LTSA	BPELVT BPEL2STS

Table 4. Property coverage of BPEL transformation tooling

All examined other verification tools consider a BPEL process to be a concurrent system, and focus on verifying reachability, liveness and deadlock using a model checker. However, STP processes are sequential. The BPEL2CFG tool we developed was the only tool capable to verify the operation of a BPEL STP process taking data transformations into account. A comprehensive report of the results and more use cases can be found in the extended version of our paper [4]

5 Conclusion

In this paper, we developed a tool-supported hybrid analysis for sequential BPEL processes to 1) checking correctness of protocol and data properties based on parsing, 2) automatically generate testcases in the form of a traces of BPEL events that syntactically obey a given protocol. To the best of our knowledge, our approach is the first to fully support the verification the data (as well as the protocol) of BPEL processes. It is therefore complementary to existing tools, which focus mostly on performance and concurrency properties such as deadlock, liveness and timing failures. The developed tools, the full source code to reproduce the industrial case in our own tool suite and the other described tools, and an extended version of this paper with many more details can be found on our Github repository [4].

As future work we plan to develop support for higher coverage of the BPEL standard, in particular concurrency features. Furthermore, since data transformations are fairly complex to handle with the Eclipse Epsilon plug-in, we aim to investigate if reusing components of the Apache ODE engine (which is also used in run-time monitoring) can simplify the translation. Finally, it would be interesting to develop a grammar-based verification approach for BPMN, which is another XML-based process specification language widely used in industry.

References

1. F. Abouzaid and J. Mullins. A calculus for generation, verification and refinement of bpel specifications. *Electronic Notes in Theoretical Computer Science*, 200(3):43–65, 2008.

2. M. Bozkurt, M. Harman, and Y. Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.
3. M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong. Veriws: a tool for verification of combined functional and non-functional requirements of web service composition. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 564–567. ACM, 2014.
4. E. de Jager and S. de Gouw. Github repository with extended paper and all tools and artifacts. *https://github.com/erwindejager/ou/tree/master/archive*.
5. C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.
6. D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.
7. N. Kokash, C. Krause, and E. P. de Vink. Data-aware design and verification of service compositions with Reo and mCRL2. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2406–2413, 2010.
8. C. Luo, S. Qin, and Z. Qiu. Verifying bpel-like programs with hoare logic. *Frontiers of Computer Science in China*, 2(4):344–356, 2008.
9. S. Morimoto. A survey of formal verification for business process modeling. In *International Conference on Computational Science*, pages 514–522. Springer, 2008.
10. T. Parr and K. Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of PLDI 2011*, pages 425–436, 2011.
11. F. C. Pereira and D. H. Warren. Definite clause grammars for language analysisÃćÂĂÂŤa survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980.
12. O. Standard. Web services business process execution language version 2.0. *URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*, 2007.
13. W. M. Van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance checking of service behavior. *ACM Transactions on Internet Technology (TOIT)*, 8(3):13, 2008.