Operational Semantics for BPEL Complex Features in Rewriting Logic (work in progress)*

Joseph C. Okika[1], Olaf Owe[2], and Cristian Prisacariu[2]

[1] Department of Computer Science, Aalborg University,
Aalborg, Denmark
E-mail: ojc@cs.aau.dk

[2] Department of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mail: {olaf,cristi}@ifi.uio.no

1 Motivation

Business Processing Execution Language (BPEL) is a core part of the Web Services protocol and a service orchestration language for Service Oriented Architecture (SOA). Formal semantics of full BPEL is under intense research in the SOA community because of the inherent difficulties of some complex (and important) features of BPEL. These features include the nested concurrency model and the combination of communication, exception handling, and shared resources. Apart from the common concurrency problems, such as deadlock or race condition, one needs to consider that BPEL supports developing applications with multiple services running in parallel and equipped with a transaction mechanism called compensation. Further, a service in this setting does not have a full knowledge of services that it will interact with in advance. These lead to synchronization problems, e.g.: how do different services synchronize on data? How are faults (exceptions) in nested concurrency handled?

The efforts invested into finding a formal semantics for BPEL have several motivations: a formalization of BPEL would provide better understanding of the language, and would facilitate building of tools for simulation, analyses, and verification of systems with mathematical precision. Among the formalizations found in the literature, those based on Petri nets offer support for simulation [3, 6], whereas those based on labeled transitions systems offer support for automatic verification [2, 7, 4]. However, because the complex features of BPEL are different from the conventional programming languages, these formalizations do not fully capture the meaning of these features.

In our work we take an operational semantics approach and formalize some of the complex features of BPEL based on rewriting logic [5]. The advantage of this approach is that the semantics is now executable in Maude [1]. This caters for analyses of BPEL processes using simulation and for verification using breath-first search.

* Partially supported by the Nordunet3 project "COSoDIS – Contract-Oriented Software Development for Internet Services" (http://www.ifi.uio.no/cosodis/).

$$
\begin{array}{lll}
concurrency & ::= & flow\ \mathcal{A}_1\ \mathcal{A}_2\ \dots\ \mathcal{A}_n \\
compensation & ::= & compensate\ s \mid compensateScope\ s \\
scope & ::= & scope\ decl\ s\ \mathcal{A}\ eventhandler^*\ faulthandler^* \\
 & & compensationhandler\ terminationhandler \\
bproc & ::= & decl\ eventhandler^*\ faulthandler^*\ \mathcal{A} \\
eventhandler & ::= & p\ \mathcal{A} \mid d\ \mathcal{A} \\
faulthandler & ::= & F'\ \mathcal{A}\ \text{where}\ F' \subseteq F \\
compensationhandler & ::= & \mathcal{A} \\
terminationHandler & ::= & \mathcal{A}
\end{array}
$$

Table 1. Abstract Syntax for concurrent flow, process, scope and handlers

2 Complex Features of BPEL

A BPEL process models the public behavior of interacting partner services or the private behavior of a partner service in a business interaction. It specifies when to wait for messages, and when to send messages among other activities. BPEL processes are programmed using the following constructs:

Declarations of process variables, partner links, correlation sets, etc. are done at a process level, in addition to handlers (without compensation and termination). For a scope, all scope declarations apply at the scope level.

Scope declares variables similar to a process including compensation and termination handlers. It provides a context for processing these declarations and activities.

Handlers (<eventHandlers>, <faultHandlers>, <compensationHandler>, terminationHandler) can be defined for the process or for a scope to manage exceptions.

Compensation is used to manage long-running transactions by reversing successfully completed activities. Compensations can be activated by <compensate> and <compensateScope> activities.

Activity includes the basic activities (<assign>, <validate>, <empty>, <throw>, <rethrow>, <wait>) for data manipulation and service interaction; the structuring activities (<sequence>, <while>, <if>, <repeatUntil>, <forEach>) for organizing activities; and the concurrent activity (<flow>) for capturing concurrency.

Among all these, the more complex and important ones that we deal with are listed in Table 1 (in abstract syntax). The $flow$ construct captures the concurrency aspect of BPEL. The other constructs are for *scope* and various handlers for managing exceptions. An event handler can be a message based event (receiving a message on a specified port p) or an alarm based event (timeout of a specified time d). In the case of the fault handlers there can be a specific fault (handled by a *catch* clause in BPEL) or a set of faults (handled by a *catchall* clause) hence the $F' \subseteq F$ in the syntax definition, where F is the set of fault names, $\mathcal{A}$ denotes the set of activities, and s a scope name.

3 Operational Semantics Approach using Rewriting Logic

Rewriting logic and its tool Maude provide a semantical framework and a high-level language for defining operational semantics for languages; and have been used to formalize a number of modeling languages and paradigms. The facilities for prototyping and tool support are valuable when experimenting and comparing different semantic definitions. An advantage of rewriting logic is that it has a general, built-in notion of concurrency and readily supports different mechanisms for interaction, including object-oriented systems and message passing systems. Concurrency is typically based on defining multi-sets of units, which become concurrent when the left hand sides of the rewriting logic rules concern only one unit. Conceptually, several units are rewritten at the same time by applying the rules to all of them independently.

In order to formalize BPEL one needs the notion of nested concurrency, embedded in nested scopes of handlers, and variable declarations. Furthermore, interaction is based on both events, exceptions and shared resources. These features call for a a very general framework. In our work we will show that the Maude concurrency model can be adapted to the setting of nested systems in a natural way. A BPEL configuration consists of an environment reflecting the scope structure, including the binding of entities and the context of handlers, together with a program continuation and inner configurations representing concurrent processes at that scoping level. Event-based communication can then be formalized with true concurrency semantics, in the sense of non-overlapping rule applications, whereas shared resources give rise to non-deterministic interleaving of the actions depending on a shared resource. In this way the main challenges of BPEL are solved in a unified manner. BPEL processes can then be simulated by running the operational semantics in Maude with the BPEL processes as input. The set of all possible executions can be explored by the search command; both breath-first search and bounded breath-first search.

References

1. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.
2. X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, December 2005.
3. N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services (FABPWS'07)*, pages 21–35. University of Podlasie, June 2007.
4. R. Mateescu and S. Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In *CIAO! / EOMAS*, pages 179–193, 2008.
5. J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.
6. C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.
7. G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the Semantics and Verification of BPEL4WS. *Electr. Notes Theor. Comput. Sci.*, 151(2):33–52, 2006.