A Model Checking Approach to Verify BPEL4WS Workflows

Domenico Bianculli
Faculty of Informatics
University of Lugano
via G. Buffi 13 - CH-6900, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch

Carlo Ghezzi Paola Spoletini
Dipartimento di Elettronica e Informazione
Politecnico di Milano
via Ponzio 34/5 - I-20133, Milano, Italy
{ghezzi, spoleti}@elet.polimi.it

Abstract

The increasing diffusion of service oriented computing in critical business transactions demands reliability and correctness of the workflow logic representing web service orchestrations.

We present an approach for the formal verification of workflow-based compositions of web services, described in BPEL4WS. Workflow processes can be verified in isolation, assuming that the external services invoked are known only through their interface. It is also possible to verify that the actual composition of two or more processes behaves correctly. We can verify deadlock freedom, properties expressed as data-bound assertions written in WS-CoL, a specification language for web services, and LTL temporal properties. Our approach is based on the software model checker Bogor, whose language supports the modeling of all BPEL4WS constructs.

We provide an empirical evaluation of our approach and we compare the results with other BPEL4WS model checking tools.

1. Introduction

In recent years, Web Services (WSs) have been proposed as a practical approach to the development of distributed applications supporting virtual information systems for networked enterprises. Numerous technologies have been developed to support WSs and standardization efforts have been initiated, with substantial industrial support. More generally, there has been a growing interest in Service Oriented Computing (SOC) and Service Oriented Architectures. SOC can be defined as an approach to distributed computing that views software resources as dynamically discoverable services available on the network, supporting interoperability of heterogeneous applications and infrastructures.

The spreading of WSs and SOC in several application domains and their use in increasingly critical settings ask for a level of dependability that current approaches cannot assure. The verification and validation techniques available for traditional software must be adapted and extended to deal with the new problems arising in these cases.

Considerable attention has been recently drawn by model checking as a practical approach to software verification. Although the idea of formally verifying software has been around for decades, it failed to become common practice. Model checking is an automatic approach that may indeed favor this transition. Model checking [10], born as a practically efficient technique for automated verification of hardware systems, can also be a good candidate for verification and validation in the context of SOC. A model checker, provided in input with a model of a system and a certain desirable property, verifies if the system satisfies the property; if the property does not hold, the model checker reports a counterexample, showing an execution where the property fails.

The application of model checking to SOC raises numerous research challenges, because systems are intrinsically distributed and the environment in which they run has to be taken into account (i.e. modeled as well).

The work described in this paper focuses on verifying WS compositions through a workflow (also called *orchestrations*). Workflow-based compositions provide new services by composing existing services. We need to verify compositions because they must provide dependable services.

This paper proposes BPEL2BIR, a tool for model checking WS compositions described in the BPEL4WS [1] workflow language, which has become the de-facto standard for defining WS orchestrations. Model checking is performed by using the Bogor model checker. Our approach provides full coverage of BPEL4WS constructs and generates very

compact models. Additionally, it offers the possibility of analyzing stand-alone BPEL4WS processes as well as compositions of web-based processes. These features make our approach an improvement over previously published works that deals with BPEL4WS verification, as we discuss in Sect. 3.

The paper is organized as follows. Section 2 provides a brief description of the Bogor model checker. In Section 3 we survey related work. In Section 4 we illustrate our approach for model checking BPEL4WS workflows with Bogor and then in Section 5 we discuss experimental results. Finally, in Section 6 we draw some conclusions and present guidelines for future investigations.

2. The Bogor Model Checker

Bogor [29] is a model checking framework developed at Kansas State University. The input language of Bogor is called BIR, Bandera Intermediate Representation. BIR provides constructs found in modern programming languages, such as dynamic thread and object creation, exception handling, virtual functions, recursive functions and garbage collection. Moreover, a low-level version of the intermediate representation, named low-level BIR, consists of a guarded command transition system language with explicit locations, explicit guards, sequence of statements comprising a transition action, and explicit transitions.

The BIR data model contains both primitive types (such as `boolean`, `int`) and non-primitive types (`record`, `array`, `lock`). For record types, sub-type declarations and virtual methods are also supported. The language is statically strongly typed. The memory model disallows pointer arithmetic and supports object reclamation through garbage collection.

To overcome the *state explosion* problem [10], typical of model checking, Bogor implements some well known optimization/reduction strategies, such as data and thread symmetry [12], collapse compression [30] and partial order reduction [16].

Bogor's main feature is *extensibility*, both in terms of input language and model checking algorithms. It has an open, modular architecture that supports the development of extensions via new algorithms and optimizations, to improve core tasks, such as state-encoding, and state-exploration. Several extensions have been implemented. Besides partial-order reduction, state-encoding and different search strategies, there are extensions to support different property languages (regular expressions, LTL and CTL, JML), several domains (multi-threading and Swing libraries, event-based Java programs, CORBA-based avionics systems). In our group we also designed Bogor extensions to model event-based service-oriented architectures based on the Publish/Subscribe paradigm [6].

3. Related Work

Research on formal verification of web services is quite recent, but has attracted considerable attention. In this section we review the state of the art in verification, focusing on model checking approaches.

An initial attempt has been described in [25]. Flows are described in WSFL [24] and translated into Promela, the input language of the SPIN model checker [20]. By following this approach, one can verify reachability, deadlock-freedom and application-specific properties.

WSAT, a framework for analyzing interaction of composite web services is presented in [14] and [15]. The interactions of composite web services are modeled as conversations, keeping track of exchanged messages. BPEL4WS specifications of web services are translated into an intermediate representation, an XPath-guarded automaton augmented with unbounded queues for incoming messages. This model is then translated into Promela and LTL properties, which can be also derived from XPath expressions, are checked with the SPIN model checker.

The Zing model checker [2] is reported[1] to check for errors in sets of web services, whose behavior is described in BPEL4WS.

The Verbus verification framework [4] is a modular and extensible framework for the verification of business processes. Thanks to an intermediate formalism, the framework is not tied to specific process definition languages or verification tools. The support for the BPEL4WS language is partially complete: missing constructs are the *compensation* activity and event handlers. The current version of the prototype performs activity reachability analysis and supports verification of such properties as invariants, goals, activity pre- and post-conditions, as well as generic properties defined in temporal logic.

Nakajima [26] proposes a method to extract the behavioral specification from a BPEL4WS process and to analyze it by using the SPIN model checker. A finite state automaton extended with variable annotations (definitions and updates) is used as an intermediate representation. This approach provides only a partial support for BPEL4WS, which does not deal with fault/event handlers and compensation activities. The tool checks for deadlock freedom and verifies user-defined LTL properties.

In [28], an operational semantics is provided for a subset of the BPEL4WS language, which is then mapped onto a network of Timed Automata, verified using Uppaal [8].

Other authors use different computational models for verifying BPEL4WS processes. Petri Nets are used in [31], where a Petri Net semantics is provided for BPEL4WS. The

[1] According to the presentation available at `http://research.microsoft.com/zing/zing-overview.ppt`.

net resulting from the translation is then validated with the LoLA [32] model checking tool.

Process algebras are used in [13] and [22]. In [13], web service compositions are verified against properties created from design specifications and implementation models. Specifications, in the form of Message Sequence Charts, and implementations, in the form of BPEL4WS processes, are translated into the Finite State Process notation, which is the input language for the LTSA (Labelled Transition System Analyzer) model checker. In [22], a process algebra, the BPE-calculus, is used to abstract BPEL4WS control flow. This calculus is used as input for a process algebra compiler to produce a front-end for the concurrency workbench (CWB) [11], in which equivalence checking, preorder checking and model checking of processes are performed.

The work described in [21] proposes model checking of composite web services expressed with OWL-S (Web Ontology Language for Web Services) [3], using an extended version of Blast [19] tuned to support OWL-S concurrency. Atomic web services are verified for predicate-bound properties.

4. Model Checking BPEL4WS with Bogor

Web services can be used as components to build highly decentralized and evolvable system architectures [5]. Web services live in an open world, they normally belong to different administrative domains, and may become available dynamically. In such a case, the services retrieved from the open environment and composed to build a higher-level service must be treated as black boxes. Their internal behavior is not visible externally. The workflow process only knows such services through their interface specification, which describes their expected behavior. In such a case, we can only verify a workflow as a stand-alone process, i.e. we perform *intra-service* verification, where external services are abstracted by their specification. There are cases, however, where we are allowed to open the black box. This may happen, for example, in the case of services developed within the same department or consortium, or in the case of open-source services. In such cases, the dynamically published services can be viewed as glass boxes, which expose their internals. As a consequence, verification can take advantage of the detail information that becomes available as an external service is viewed as a glass box. It is thus possible to achieve *inter-services* verification, by formally analyzing properties of service compositions.

This dual-mode approach can be exploited for the verification of *local* and *global* properties. Indeed, in the first case one may be interested only in verifying data-bound or reachability properties within the workflow of a single process. In the latter, verification may focus on the behavior of the whole composition, for example by proving that a certain temporal property on the exchange of messages among business partners holds.

Table 1. Comparison of BPEL4WS constructs support

BPEL4WS constructs	BPEL2BIR	Verbus[a]	Nakajima[b]
basic + structured activities[c]	yes	yes	yes
fault handler	yes	yes	no
event handler	yes	no	no
compensation handler	yes	no	no

[a] Data taken from [4].
[b] Data taken from [26].
[c] activities described in §11 and §12 of [1].

The choice of using Bogor in our approach comes from the fact that its high-level input language allows the modeling of all constructs of BPEL4WS, providing a better support for the language with respect to other similar tools, as reported in Table 1. Consider also that while our tool supports both intra-service and inter-services verification, [4] and [26] support only the former, and [14] supports only the latter, although it also supports all BPEL4WS constructs. Furthermore, Bogor's plugin-based architecture allows the exploitation of several model checking techniques and the customization for a particular domain. Although in this work we report on the use of only the basic set of features offered by Bogor, the encouraging results obtained (discussed in Section 5) suggest that we might develop BPEL4WS extensions for Bogor, as we outline in Section 6.

Before illustrating the translation, we summarize the main aspects that characterize our approach with respect to existing ones :

- it supports the analysis of both a stand-alone BPEL4WS process and of a composition of web-based processes;
- it supports the specification and verification of properties described in WS-CoL and in Linear Temporal Logic (see Section 4.2);
- it covers all of BPEL4WS constructs (except those dealing with time);
- it uses a novel extensible model checker (Bogor);
- it offers significant efficiency gains, in term of the size of the model, over previous verification systems.

The remaining parts of this section are structured as follows: Section 4.1 describes the translation from BPEL4WS to a BIR model and Section 4.2 details some aspects of the verification.

4.1. From BPEL4WS to BIR

BPEL2BIR implements a translator which accepts as input a set of WSDL [33] and BPEL4WS files and outputs a BIR model. The auto-generated BIR file can be then verified using Bogor.

A BPEL4WS process is mapped to a composition of threads, where a thread models the main control flow of the process and the other threads correspond to the activities nested in the `flow` constructs of the process (more on this later).

If we represent a service in isolation, the set of threads are represented as a `system`, i. e. a Bogor verification unit, and the thread that models the main control flow of the process is `active`. Instead, if we represent a composition of services, the whole composition is a `system`, hence the threads representing the main control flow of each process are not active. In fact, the system has only one active thread that just launches the main thread of each process in the composition.

In general, a system contains:

- declarations of data types, which model the message types handled by the process;
- declarations of exception types, which model faults in the process;
- global variables, for modeling the data flow of the process;
- helper variables (e.g. boolean flags), functions and threads, supporting the translation scheme from BPEL4WS to BIR.

Simple WSDL types, such as XML Schema [34] basic types, are mapped to corresponding BIR primitive types; structured WSDL messages are mapped to record types.

BPEL4WS constructs that are concerned with external partners and do not modify the state of the process are not relevant for the translation into BIR, according to the intra-service view. Literally, they are:

- `partnerLink`, which specifies the partner from/to which a message is received/sent;
- `reply` *activity*, which models sending messages to partners;
- *correlation set*, which associates the right instance of the process with the partner that initiated it;
- *compensation*: compensation handlers operate on the saved state to perform an *undo* operation with respect to other partners.

Instead, when the inter-services view is considered, the `partnerLink` construct plays an important role since it defines the communication channel between two workflow processes. Each communication channel in one direction of a link is represented by a data variable of the same type of the values exchanged along the channel, and a boolean variable acting as a semaphore for read/write access.
Also compensation assumes an important role in this view, since compensation actions mainly involve external services. Compensation is translated by saving the pre-status of the involved variables. In more detail, in a pre-processing phase, all the elements embedded in a compensation are listed and subsequently they are represented with variables that remember their previous status. When a `compensate` occurs, the recovery action with respect to other partners is performed using old values.

The activities of a BPEL4WS process are mapped into BIR as follows.
The `assign` activity is modeled by the BIR assignment operator. The `pick` activity is translated by invoking a function that models the occurrence of one of the events being awaited; the function call sets a flag, which is then evaluated in an `if...then...elseif...else` statement to execute the body associated with the event type. The blocking nature of the activity is not modeled.
Basic BPEL4WS control flow statements such as `sequence`, `switch`, `while`, `throw`, `empty`, `terminate` are trivially mapped to their equivalent in BIR.

We model each `wait` activity in a simplified way through a boolean flag. Since we do not model the time passing, execution of the `wait` activity is modeled by setting the flag to true. This abstraction limits the use of `wait` in the `flow`, since one cannot establish an order relation among `wait` activities contained in the `flow`.

A `flow` activity $\texttt{flow}_i$ is translated into the invocation of the function $\texttt{launchAndWaitFlow}_i$, which creates and starts a thread for each activity in the parallel flow, and returns to the caller only when all the launched threads terminate. The first actions performed by each thread correspond to the initial activity of each branch of $\texttt{flow}_i$. The other actions in the thread can depend on the value of their incoming links, encoded in BIR by boolean variables corresponding to link transition conditions. In each thread, the activities that are target of some links wait for the completion of the activities they depend on and then evaluate the conditions carried by the links, using the join condition. Each thread terminates by setting the variables corresponding to the transition condition of its outgoing links and calling the `exit` function.

The `receive` (`invoke`) activity is translated in two different ways depending on the verification viewpoint. If we consider an intra-service view, the `receive` (`invoke`,

BPEL4WS pseudo-code	BIR translation

receive(price);
if (price < 10000) then
  if (price < 5000) then
  --perform activity A
  else
  --perform activity B
else
//perform activity C
	boolean price-less-10000;
	boolean price-less-5000;
	choose
	  when <true> do price-less-10000:=true;
	  when <true> do price-less-10000:=false;
	end
	if (price-less-10000) do
	  choose
	  when <true> do price-less-5000:=true;
	  when <true> do price-less-5000:=false;
	  end
	  if (price-less-5000) do
	  //translation of A
	  else do
	  //translation of B
	else do
	//translation of C

Figure 1. Example of variable abstraction

respectively) is translated as an assignment to its input (respectively, output) variable, since the behavior of external services is not modeled. The assignment is performed with a value either extracted non-deterministically from the domain (in the case of boolean or restricted integer variables) by a "generative" code block, or provided by the users. Alternatively, depending on the properties they want to verify, users may also specify that a variable may be abstracted: i. e., each expression using a variable is substituted with a boolean predicate, assigned non-deterministically. This transformation is performed by applying a standard define-use chain analysis [27]. Figure 1 shows an example of abstraction, where the uses of variable `price` generate two predicates, represented by the variables `price-less-10000` and `price-less-5000`. When the inter-services view is considered, the details of the communication are known. In this case, the `receive` activity (`invoke` and `reply`, respectively) is translated using a guarded reading access (assignment, respectively) to the channel data variable, with the guard being the channel semaphore.

BIR does not support nested blocks; local variables of nested BPEL4WS scopes are declared as local variables of the thread in which the scope is translated. However, a `try/catch` block is generated to model the local fault handler.

Each fault handler in a scope of a BPEL4WS process is associated with a specific fault; we model this feature of the language by declaring a new variable —named after the fault name— of type `throwable record`, and by appending to the `try` statement in the current scope a new `catch(var)` clause matching the exception variable `var` corresponding to the fault.

The notification of external events is modeled by introducing in the model a helper function, the `eventGenerator`, which non-deterministically produces constants representing timer or message events. When the first statement of a scope is entered, a new thread is started. This thread, which runs until the last statement of the scope is executed, retrieves a new event by invoking the helper function. If the returned value matches one of the messages declared in the event handler associated with the scope, the thread executes the corresponding activity. The translation details of non-trivial constructs are summarized in Table 2.

4.2. Specifying and Verifying Properties

If no user-specified properties are provided, Bogor checks the automatically generated model for deadlock freedom. In our approach, users may also specify additional properties using WS-CoL and Linear Temporal Logic.

WS-CoL is an assertion language defined in [7] for specifying monitoring expressions; it is based on JML [23], with some conceptual and syntactical differences due to the adaption to the world of web services. WS-CoL allows the designer to predicate on variables containing data originating both within and outside the process. It uses predefined variable functions depending on the variable's data type, it combines basic constraints through the use of typical boolean operators and uses universal and existential quantifiers. As opposed to JML, WS-CoL does not support the keywords `\old` and `\result`. They are not needed because service invocations cannot modify input parameters and because returned messages can be referred by their names.

An example of a WS-CoL property is the following: `($msg/amount) > 0 && ($msg/amount) <=4`, which predicates over the variable `msg`, whose type is a structured WSDL message having `amount` as a field of type integer.

In our approach WS-CoL properties can be directly em-

Table 2. Translation scheme from BPEL4WS to BIR

name	BPEL4WS syntax	BIR syntax

pick	<pick ...>
	<onMessage ...>
	activityA
	</onMessage>
	<onAlarm ...>
	activityB
	</onAlarm>
	</pick>
		enum Pick1_T {message1, alarm1};
		...
		Pick1_T flag1;
		...
		flag1:= pick1();
		if flag1 == Pick1_T.message1 do
		activityA:
		else do
		activityB;
		end

flow	<flow ...>
	activityA
	activityB
	</flow>
		tid flow1_act1_tid;
		tid flow1_act2_tid;
		...
		launchAndWaitFlow1();
		...
		function launchAndWaitFlow1() {
		boolean temp0;
		loc loc0: do {
		flow1_act1_tid := start flow1_act1();
		flow1_act2_tid := start flow1_act2();
		} goto loc1;
		loc loc1: do {
		temp0:=threadTerminated(flow1_act1_tid)
		&& threadTerminated(flow1_act2_tid);
		} goto loc2;
		loc loc2: when temp0 do{} return;
		when !temp0 do{} goto loc1;
		}
		thread flow1_act1() {activityA; exit;}
		thread flow1_act2() {activityB; exit;}

fault	<faulthandlers> <catch faultName="x:a"> activityA </catch> <catch faultVariable="x:b"> activityB </catch> </faulthandlers>	throwable record A; throwable record B; ... try ... catch (A a) ... catch (B b) ... end

bedded in the XML code of the BPEL4WS process and then translated as `assert` statements in the BIR model.

We also support the verification of properties specified as linear temporal logic formulas, embedded in the XML code of the BPEL4WS process, and then translated in the BIR model. LTL verification is performed by using two extensions of Bogor, namely `property-ltl` and `property-buechi`.

5. Experimental Results

We assembled a test suite of BPEL4WS processes that we used to evaluate the effectiveness of our approach and thus the "quality" of the automatically generated code, from a model checking point of view.

Before defining our own test set, we checked whether we could adopt some existing standard benchmark set of workflows[2]. Since, unfortunately, such a benchmark does not exist yet, we decided to build our own by assembling the published experiments that were used for the verifiers we compare to.

Our suite is composed of three packages : 1) two examples coming with the Verbus distribution[3]; 2) the four examples described in the BPEL4WS standard; 3) the BPEL4WS example of service composition included in the WSAT distribution[4].

The first two packages refer to intra-service verification, while the third deals with inter-services verification. Hence, the first two packages allow us to compare our tool with the Verbus framework and Nakajima's approach [26], while the third supports a comparison with WSAT.

For the first and the third experiment, we used the numerical constants and integer domains specified in the files of Verbus and WSAT distributions, respectively; for the sec-

[2]Such as a benchmark exists in other areas. For example, Generalized Railroad Crossing problem [18] is a proposal for specifying and verifying real time systems.

[3]Available at `http://www.it.uc3m.es/~jaf/verbus/`.

[4]Available at `http://www.cs.ucsb.edu/~su/WSAT/`.

Table 3. Comparison of outputs

Framework	Process	States	
		Bogor	SPIN
Verbus	Olive	932	10508
	Orders	27	324
Nakajima	Purchase	24	2497
	Shipping	22	216
	Loan	1625	3516
	Auction	24	57
WSAT	Loan + LTL	200181	980324

ond experiment, we used variable abstraction, as in [26]. In the third experiment, we considered the `LoanApproval` example and we verified the following LTL property: $\Box(\Diamond(\text{approvalMsg} = \text{'yes'} \vee \text{approvalMsg} = \text{'fault'}))$, which states that the loan approval process eventually will reply either with an output message or a fault.

Experimental analysis of a prototype implementation of BPEL2BIR has been performed on a machine with a 1.7GHz Intel Pentium M processor and 512 MB RAM, running GNU/Linux; we used Bogor ver. 1.2.20060221 and SPIN ver. 4.2.6.

The results of the experiments summarized in Table 3 show that in all cases our translation method provides a more compact model, in terms of number of states, with respect to other approaches. We do not provide memory usage statistics since the value reported at the end of the verification by Bogor is inconsistent[5], due to the lack of good memory control/query in Java and the activity of the garbage collector.

6. Conclusions and Future Work

In this paper we presented an approach to formal verification of business processes, described in BPEL4WS, using the Bogor model checker. Bogor's high-level input language allows the modeling of all constructs of BPEL4WS relevant for model checking the workflow. Our initial experiments show the validity of our approach, not only in terms of coverage of the language, but also in terms of performance of verification, because of the smaller size of the generated models. A more thorough empirical evaluation of the existing approaches, however, is still necessary to fully understand the practical applicability of formal verification via model checking. This will be one of our future research directions.

Our future work will also address two different, orthogonal, directions. On the modeling side of the approach, we will add further support for the `wait` activity embedded in `flow` constructs. Moreover, we will provide a reverse-mapping from BIR to BPEL4WS, which will be useful to show counterexample traces directly in the BPEL4WS source code, instead of low-level BIR, as we currently do. Furthermore, the definition of a formal semantics of both BPEL4WS and BIR could help in proving a bi-simulation between the two models, to formally assess the correctness of our approach, which so far has only been informally derived from the proposed translation.

As for the model checking theory's side, we plan to exploit Bogor's extensibility for a further development. The CEGAR (Counterexample Guided Abstraction Refinement) [9] loop and predicate abstraction [17] state space reduction techniques — which proved to be highly beneficial when applied to software model checking — may be implemented as Bogor plugins to improve verification efficiency.

Last, since our work is based on version 1.1 of BPEL4WS, we plan to support the forthcoming release of the language as soon as it will become the official standard.

[5] As reported in `http://projects.cis.ksu.edu/forum/forum.php?thread_id=308&forum_id=16`.

7. Acknowledgments

Part of this work has been supported by the IST EU project "PLASTIC" — contract number 026955 — and the italian FIRB project "ART DECO".

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Technical report, OASIS, 2003-05-05 2003.

[2] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[3] A. Ankolekar. OWL-S Semantic Markup for Web Services, 2003. `http://www.daml.org/services/owl-s/`.

[4] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Formal Verification of BPEL4WS Business Collaborations. In *E-Commerce and Web Technologies, 5th International Conference, EC-Web 2004, Proceedings*, volume 3182 of *Lecture Notes in Computer Science*, pages 76–85. Springer, 2004.

[5] L. Baresi, E. Di Nitto, and C. Ghezzi. Towards Open-World Software: Issues and Challenges. *IEEE Computer*, 39:36–43, October 2006.

[6] L. Baresi, C. Ghezzi, and L. Mottola. Towards Fine-grained Automated Verification of Publish-Subscribe Architectures. In *Proceedings of the 26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE06)*, Paris, September 2006.

[7] L. Baresi and S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[12] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996.

[13] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)* , pages 152–163. IEEE Computer Society, 2003.

[14] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.

[15] X. Fu, T. Bultan, and J. Su. WSAT: A Tool for Formal Analysis of Web Services. In *Computer Aided Verification, 16th International Conference, CAV 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514. Springer, 2004.

[16] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[17] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[18] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Proc. Tenth Intern. Workshop on Real-Time Operating Systems and Software*. IEEE Computer Society Press, may 1993.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[20] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[21] H. Huang, W.-T. Tsai, R. Paul, and Y. Chen. Automated Model Checking and Testing for Composite Web Services. In *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*, pages 300–307. IEEE Computer Society, 2005.

[22] M. Koshkina and F. van Breugel. Verification of Business Processes for Web Services. Technical Report CS-2003-11, York University - Department of Computer Science, 4700 Keele Street, Toronto, M3J 1P3, Canada, October 2003.

[23] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[24] F. Leymann. Web Services Flow Language (WSFL) 1.1., 2001.

[25] S. Nakajima. Verification of Web Service Flows with model checking techniques. In *CW '02: Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, page 0378, Washington, DC, USA, 2002. IEEE Computer Society.

[26] S. Nakajima. Model-Checking Behavioral Specification of BPEL Applications. In *Proceedings of the International Workshop on Web Languages and Formal Methods, WLFM 2005*, 2005.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[28] G. Puyz, Z. Xiangpengy, W. Shulingy, and Q. ZongyanyS. Towards the Semantics and Verification of BPEL4WS. In *Proceedings of the International Workshop on Web Languages and Formal Methods, WLFM 2005*, 2005.

[29] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11*, pages 267–276, 2003.

[30] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proceeding of the 2003 Workshop on Software Model Checking*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003.

[31] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3–26, March 2005.

[32] K. Schmidt. LoLA: A low level analyser. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag, June 2000.

[33] W3C. Web Services Description Language (WSDL) 1.1, 2003. `http://www.w3.org/TR/wsdl`.

[34] W3C. XML Schema, 2004. `http://www.w3.org/XML/Schema`.