# CHAPTER 1

# INTRODUCTION: SERVICE RELIABILITY



Service-oriented architecture (SOA)–based applications are built as an assembly of existing web services wherein the component services can span across several organizations and have any underlying implementations. These services are invoked in a sequence based on the business logic and on the workflow of the application. They have alleviated software interoperability problems and catapulted SOA into the forefront of software-development architectures. The rapid inroads made by such applications can be attributed to their agility, maintainability, and modularity.

SOA-based applications require software components to be exposed as *services*. Each service has an *interface* that outlines the functionality exposed. Ideally, an application is designed by discovering appropriate services, using their interfaces, and composing them. Such static compositions require the services involved to be perpetual and consistent throughout the lifetime of the application. Microsoft Biztalk [52] and Oracle WebLogic [38] are among popular static composition engines.

However, existing web services can break, and newer (and probably better) services can surface. Furthermore, a change in the business logic of an application during its lifetime might necessitate that additional web services be composed dynamically. Such a state of events has culminated in dynamic web service composition. Compared to their static counterparts, an application based on dynamic

*Verification of Communication Protocols in Web Services: Model-Checking Service Compositions*, First Edition. Zahir Tari, Peter Bertok, Anshuman Mukherjee.

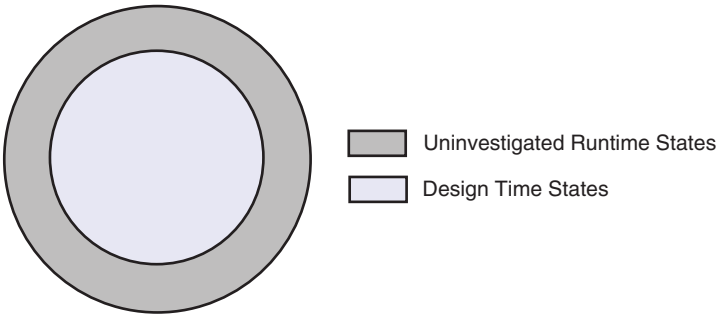


**FIGURE 1.1** Additional states for runtime verification.

composition is open to modification, extension, and adaptation at runtime. Stanford's Sword [44] and HP's eFlow [7] are among the popular dynamic service composition platforms.

Nevertheless, dynamic composition [9,57] presents immense challenges. An enterprise application is expected to be void of deadlocks, live-locks, and conflicts. A static composition can be *verified* for these behavioral properties at design time. However, the verification of dynamically composed applications can be done only at runtime. This is exacerbated further by services that do not conform to their published interfaces at runtime. Considering the remarkable ingenuity of formal methods in runtime verification of traditional systems, they ought to be used for SOA-based applications [5,28].

The reliability and usability of SOA-based applications require lifelong verification of the corresponding service composition. This includes verifying a composition at design time and monitoring its behavior at runtime. Verification at the design stage involves generating and scrutinizing the entire state space of the composition for behavioral properties. Unless the underlying composition is altered at runtime, the application would always be in one of the states scrutinized. However, as shown in Figure 1.1, the application might reach uninvestigated states at runtime, owing to the dynamic nature of the composition. These states could be reached if services in the target application are added, removed, or updated. To verify the behavioral properties for runtime states, the model checking should not terminate with design-time verification. Instead, it should continue at runtime to determine the uninvestigated states reached by the application and to scrutinize them to verify the behavioral properties.

Conventional techniques [1,42] cannot be used to verify SOA-based applications, for several reasons. First, most faults are related to the business logic of the service composition rather than to the source code or implementation of underlying services; second, even if an issue has been found with the implementation of a service, the source code is usually not available for rectification; and third, even if the source code is available, it cannot be rectified immediately, as this might break many other applications using this service. Furthermore, the reliability of conventional verification methods were seriously undermined by the *Ariane 5* rocket launch failure [15] and the deaths due to malfunctioning of a Therac-25 radiation

therapy machine [45] despite rigorous software testing. The teams investigating these disasters recommended using formal methods to complement testing, as the former assures exhaustive verification of a system [15,45].

Formal methods have remarkable ingenuity in warranting the safety and usability of a system. They involve writing a formal description of the system that is under deliberation and analyzing it to discover faults and inconsistencies. The formal description of a system is abstract, precise, and complete [33]. Although the abstractness allows a high-level understanding of the system, all inconsistencies and ambiguities in it are resolved in formulating a precise and complete description. Furthermore, the abstractness makes it possible to ignore the underlying architectural differences in SOA-based applications and to analyze them like any other software application. At the early stages of application development, formal methods are often employed that involve requirement analysis, specification, and high-level design.

The formal description of SOA-based applications should comprise the business logic for underlying service composition. Among all the domain-specific languages that were proposed for specifying web service composition, the Business Process Execution Language for web services (BPEL4WS or simply BPEL) [2,4,17] stands out as the de facto industry standard. Unfortunately, the overlapping constructs [54] and the lack of sound formal or mathematical semantics [46,51] in BPEL do not allow it to be used as a formal description. These inconsistencies are the outcome of two conceptually contrasting languages (Web Services Flow Language (WSFL) [37] of IBM and XLANG [50] of Microsoft) that were amalgamated to constitute BPEL [46]. This necessitates transforming the textual specification of BPEL into a formal description prior to formal analysis.

Unfortunately, existing solutions for formalizing a BPEL specification are ad hoc and temporary [26,40,55,56]. Despite the many modeling languages available (e.g., Promela, petri nets, automata, process algebras), these solutions specifically target a particular language. In pursuit of a generic solution, we transform a BPEL specification into an intermediate specification before the actual formalization. In software engineering, data transfer objects (DTOs) constitute a commonly used design pattern for storing and transferring data [16], and we use DTOs to store the generic intermediate specification, wherein each BPEL activity is mapped to a separate DTO. These DTOs can then be transformed into any modeling language.

However, model-checking techniques often have associated time and memory requirements that exceed the resources available, and software developers often skip formal methods because of budget and deadline constraints. Consequently, it is necessary to address these issues before we can use the method to verify SOA-based applications. A BPEL specification verification is preceded by memory- and time-reduction procedures for model checking [15]. The time and memory costs for model checking are linked—any reduction in memory requirements entails an increase in execution time [22].

Although it is possible to check a system model irrespective of its size and orientation (i.e., flat or hierarchical), it is important to recognize the advantages of a hierarchical and succinct system representation. A hierarchical model is easy to draw, analyze, and maintain [39]. Analyzing a system model might also assist in

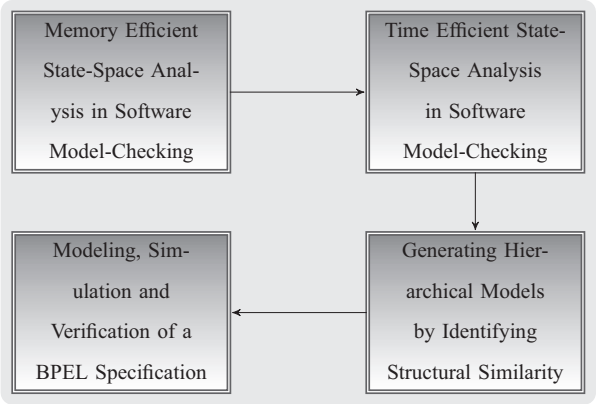


**FIGURE 1.2** Verification steps.

accomplishing additional objectives, such as identifying the overall architecture of the system, understanding its dependencies, visualizing the flow of information through it, identifying its capabilities and limitations, and calculating its complexity [13]. Having recognized the importance of concise and modular system representation, we present techniques to introduce hierarchy into a flat model. As illustrated in Figure 1.2, this technique immediately precedes the procedure for model checking an SOA-based application.

## 1.1 MOTIVATION

SOA-based applications are prone to failures and inconsistencies, owing to multiple *single points of failure* (SPOFs) [8,36]. An SPOF is an element or part of a system whose failure leads to a system crash and eventually to the termination of service or to erroneous behavior. Typical SPOFs in a service composition could be (1) the composition itself, and (2) any component service that is also created by composing services. This is essentially because the reliability of the constituent services does not guarantee the reliability of the composition. Consequently, irrespective of efforts to validate the individual services, their composition remains an SPOF. The problem is further exacerbated by dynamic service compositions wherein services can be added to, removed from, or replaced in the composition at runtime. Considering that an SOA-based application constitutes a hierarchy of services, a failure at any level can break the application. This is illustrated in Figure 1.3.

Although model-checking techniques can be used to verify a service composition exhaustively and determine the SPOFs, they are used sparingly, due to the associated time and memory requirements. Model checking involves scrutinizing the reachable states of a system to identify predefined undesirable properties. However, modern software systems have very large state spaces, owing to their complex and concurrent components. Consequently, the time required to examine each of these

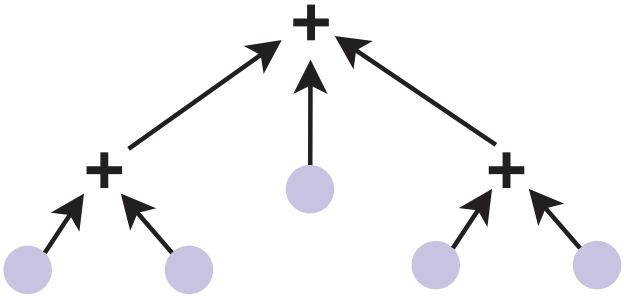

**FIGURE 1.3** An SOA-based application constitutes a hierarchy of services.

states is substantial. This is aggravated further by systems that reach one or more states repeatedly during their course of execution. Without detecting duplicate states, model checking can take a very long time. By storing the states generated hitherto in memory and comparing them to any state reached subsequently, we can reduce model-checking time significantly. However, storing the large state space of a system has significant memory requirements.

The formal representation (or model) of a service composition can also be used to determine the SPOFs, because a formal model is unambiguous, owing to its mathematical semantics. However, the representation can be huge for a complex composition, and it might be impossible for a human modeler to analyze and identify the SPOFs.

The last three decades have seen extensive research on model-checking techniques [31]. Most of the work has focused on state-space explosion [11], wherein an overwhelmingly large number of states need to be checked to verify a system. Although numerous techniques have been proposed [11,22,34,53], further improvements are still needed.

## 1.2 TECHNICAL CHALLENGES

The proliferation of web services over the last decade has called for improved reliability of service compositions, and addressing this with formal methods is a promising approach. The first two major challenges addressed in this book relate to a twofold problem: (1) the memory costs for storing states increase with the size of the state space; and (2) the delay in generating state space increases with the size of the state space. The remaining issues relate to the installation of hierarchy into a model and to the verification of a BPEL specification.

- *How can the memory costs of model-checking a service composition be reduced?* For verification, a composition needs to be formalized and then evaluated by a model-checking tool. The tool generates the state space of the system and scrutinizes it for undesirable properties. During state-space exploration,

a state can be generated more than once. To avoid analyzing the same states repeatedly, it is necessary to remember the states already explored by storing them in memory. This also ensures termination, a condition in which no new states could be generated. However, this can lead to storing a vast number of states and eventually to state-space explosion. This results in huge memory costs, as each new state has to be stored. We look into ways of reducing this memory requirement.

- *How can the time taken by model-checking a service composition be reduced?* State-space analysis of a service composition is performed by generating a reachability graph wherein each node corresponds to a state of the composition. However, the ever-increasing intricacies in contemporary SOA-based systems make the reachability graph contain a vast number of states. The resulting state-space explosion [11] leads to huge delays in producing and analyzing the state space. This is exacerbated further by having to store states that have been generated so far and compare each of them to all states produced previously. Regardless of the numerous algorithms proposed for efficient storage and comparison of states, there is always an associated time overhead. We look at various ways of reducing the delay.
- *How can we install hierarchy into a flat model to make it exponentially more succinct?* A service composition needs to be modeled prior to generating its reachability graph using one of several modeling languages available. This is a tedious and error-prone activity, and several techniques have been proposed to autogenerate a formal representation of the software system under consideration [10,27]. The primary objective of autogenerating a model is to produce the input for a model-checking tool; enhancing human understandability of the model rendered (e.g., by introducing modularity and hierarchy) is considered less important. However, the formal model for service composition might assist a human modeler in accomplishing additional objectives such as identifying the overall architecture of the composition, understanding its dependencies, visualizing the flow of information through it, identifying its capabilities and limitations, and calculating its complexity [13]. The flat models produced by autogenerating techniques provide little help in accomplishing these objectives. We examine various ways of introducing hierarchy into a flat model and making it exponentially more succinct.
- *How can we model, simulate, and verify a BPEL specification?* Being loosely coupled systems, the safety and reliability of SOA-based applications depend on the precision of service descriptions. Any implicit assumption or unforeseen usage scenarios can lead to undesirable consequences, such as deadlocks or race conditions [48]. Further complexities can be added by dynamic service composition, wherein services could be added, removed, or updated at runtime. The business process execution language (BPEL) is the de facto industry standard for service composition. Any inconsistencies or ambiguities in it will affect the reliability and hence the usability of the system produced. We look into ways of modeling, simulating, and verifying a BPEL specification.

## 1.3 SUMMARY OF EARLIER SOLUTIONS

In this section we sum up earlier solutions and point out their main limitations. In later chapters we examine them in more detail.

- *Memory-efficient state-space analysis in software model checking.* Extensive research on memory-efficient state-space analysis techniques has produced good results over the last three decades, but there are still issues to be solved. Most existing solutions can be classified as either (1) exhaustive storage [22,34,47], (2) partial storage [11], or (3) lossy storage [35,53] techniques. Exhaustive storage techniques compress and store each state in the state space, whereas partial storage techniques store only a subset of these states. Lossy storage techniques differ from exhaustive techniques in using compression algorithms that are not reversible. Exhaustive techniques are characterized by the compression algorithm used to encode a state (e.g., state collapsing [53], recursive indexing [34], very tight hashing [29], sharing trees [30], difference compression [43]). As indicated previously, each of these techniques has an associated time delay. One solution proposed [22] reduces the memory costs by 95% while only tripling the delay and is considered one of the best existing techniques. Partial techniques use specific algorithms (e.g., state-space caching) to determine states that could be safely deleted from memory. Exhaustive techniques are by far the most generic and are widely used. However, all these techniques are open to improvement, such as further reduction in the memory requirements for model checking and reduction in the associated time overhead.
- *Time-efficient state-space analysis in software model checking.* Time-efficient state-space analysis techniques have received less attention than have memory-efficient techniques. This possibly indicates a greater tolerance to delays in model checking. All existing solutions for reducing the time requirement for model checking can be categorized as either (1) partial order reduction [21,41], (2) symmetry-based reduction [18], or (3) modular state-space generation [12]. Partial order techniques involve determining *stubborn sets* (i.e., sets of transitions in which a transition outside a set cannot affect the set's behavior) and executing only the transitions enabled in each set. However, the problem of deciding if a set of transitions is stubborn is at least as difficult as the reachability problem [15]. The symmetry method exploits the presence of any symmetrical components in a system that exhibit identical behavior and have identical state graphs. The subgraphs of these components in the reachability graph of the entire system are usually interchangeable with some permutation of states. However, it is difficult to determine a subgraph whose permutations would produce other subgraphs (known as the *orbit problem* [14,19]). Furthermore, these techniques are void for models that lack symmetry. Modular state-space generation involves generating the reachability graph of each module independently and then composing them to generate the reachability graph

for an entire model. Although modular techniques look more promising, most existing solutions are dated [12].

- *Reducing the size of a formal model.* Most existing solutions decrease the size of a model by transforming it based on a set of proposed postulates. The transformations proposed by Berthelot [6] aim to reduce the size of a petri net model by merging two or more of its places or transitions based on certain conditions. The transformations used most frequently are implicit place simplification and pre- and post-agglomeration of transitions. These transformations have been extended for colored petri nets [20,32]. Although these transformations preserve several classical properties of nets (e.g., boundedness, safety, liveness) and reduce the number of reachable states when performing state-space analysis, the transformed model may not be equivalent to the original model. Consequently, an analysis of the reduced model would be incomplete.
- *Verification of a BPEL specification.* The existing solutions for formalizing a BPEL specification involve a transformation into (1) petri nets or colored petri nets [40,48,49,55], (2) process algebras [25], (3) an abstract state machine [23,24] or (4) an automaton [3,27]. The problem is addressed by generating a model for each BPEL activity using one of the aforementioned modeling languages. Thereafter, users are required to scan the BPEL specification and replace each activity with its corresponding formal model. Apart from being a cumbersome process, such an exercise is error-prone and time consuming. Although there are solutions that automate this translation, they do not consider BPEL's most interesting and complicated activities, such as *eventHandler* and *links* [27]. Despite being feature complete, the models obtained using Stahl's report [49] are bulky and error-prone, owing to the plain petri nets used. The abstract state-machine-based solutions are also feature-complete. However, they lack adequate tool support for simulation and verification.

## 1.4 SUMMARY OF NEW WAYS TO VERIFY WEB SERVICES

In this section we highlight the new techniques, models, and algorithms presented in this book to address the various problems described earlier.

- *Memory-efficient state-space analysis technique.* A technique is described to reduce the memory costs otherwise involved in model-checking a service composition by storing states as the difference from one of the neighboring states. Asserting that "the change in a state is always smaller than the state itself," storing the states in difference form results in a reduction in memory costs of up to 95%. Based on the neighboring state used to calculate the difference, the technique is divided further into two related models: (1) a *sequential model* stores a state as to how different it is from its immediately preceding state; and (2) a *tree model* stores a state as to how different it is from its nearest state in explicit form. The solution is based on the exhaustive storage technique

discussed earlier. The 95% reduction noted above allows model checking in a machine with only 5% of the memory needed otherwise. The advantage is thus twofold: (1) only 5% of the physical memory is required to validate the composition, and (2) as more states can now be stored in a memory of the same size, the chance of a complete state-space composition analysis is high. The compression algorithm achieves a 95% reduction in memory requirements with only twice the delay. Other solutions [22,34] incur considerably larger delays and offer significantly less memory reduction.

- *Time-efficient state-space analysis technique.* A method is also presented to reduce the time requirement for model checking a service composition. It includes formalizing the composition as a hierarchical model that consists of a set of interdependent modules. The reduction in delay offered is attributed to the concurrent exploration of all such modules in a hierarchical model and exposing the outcome using special data structures. These structures, known as *parameterized reachability graphs* and *access tables*, act as a repository of corresponding module behavior, and a module can use these data structures to determine the behavior of any other module without actually executing the module. In addition to concurrency, exposing such module behavior repositories eliminates the need to execute a module more than once and thereby helps, to reduce the delay. The dependency on a module of other modules is injected into its repository by using parameters. Later, these parameters are assigned specific values to obtain the corresponding reachability graph for the hierarchical model. The technique offers a time reduction of 86% in generating the first 25,000 markings. Other solutions [21,41] offer significantly less reduction in delay. Furthermore, the solution prerequisites (i.e., a hierarchical model) are less stringent than those of techniques that necessitate stubborn sets or symmetry in the model.
- *Technique for reducing the size of a model exponentially.* This technique allows an exponentially more succinct representation of a service composition by embracing the notion of hierarchy. A hierarchical model consists of a set of modules wherein each module represents a system component. In such a setup, the module for a high-level component refers to its underlying components via their module names or references. This avoids "blow-up" in including the actual representation of underlying components. Furthermore, the benefits increase with each additional high-level component sharing an underlying component. Consequently, the model obtained would be exponentially more succinct, owing to the notion of the hierarchy introduced. The solution establishes hierarchy after identifying the set of structurally similar components in a model. The experimental results indicate that this takes linear time, and the time also depends on the number of identical components in the model. The solution is generic and can be applied to any modeling language that defines the semantics of hierarchy and structural similarity.
- *Technique for modeling, simulating, and verifying a BPEL specification.* A verification framework is described to formalize a BPEL specification by

transforming it into an XML-based formal model. Existing solutions utilize ad hoc techniques to formalize a BPEL specification into a specific modeling language. However, the fast growth of SOA-based applications in recent years has necessitated the streamlining of BPEL formalization. In pursuit of generalizing the transformation, initially the specification is transformed into intermediate DTOs. This is done by extending the *Spring framework* to represent each BPEL activity using a *Java bean*. Spring helps to significantly automate the creation of intermediate DTOs. The framework instantiates the beans corresponding to activities in a BPEL specification and injects the dependencies to yield a *bean factory*, which contains all the information required to construct a formal model.

As mentioned previously, the generic intermediate specification can be transformed into any modeling language. However, to demonstrate the actual formalization, these DTOs are transformed into an XML-based formal model (e.g., colored petri nets (CPNs) [39]). This is done using the *Java Architecture for XML Binding (JAXB) 2* application programming interface (API), which offers a practical, efficient, and standard way of mapping between XML and Java code. However, the JAXB 2 API requires XML mapping for each BPEL activity. It uses the corresponding schemas to transform the bean factory into a formal model. Consequently, a CPN-based template is applied to each BPEL activity. Templates are part of a formal model. Given that CPN models (1) offer hierarchical semantics, (2) are visually expressive, (3) can be both simulated and verified, (4) are XML documents like BPEL specification, (5) have extensive tool support, and (6) support a built-in and user-defined data type, they are selected as the target for transformation. Furthermore, an object model is used to determine the relationship between BPEL activities. This object model is used to create Java beans corresponding to BPEL activities. In addition, the CPN templates exploit any hierarchy in the object model to reuse a parent template for its child activities after any required customization. The solution is feature-complete and extensible.

## 1.5 STRUCTURE OF THE BOOK

The remainder of the book is organized as follows:

- In Chapters 2 to 4 we provide the necessary background knowledge to ensure a better understanding of the relevant concepts. This includes a run-through of model checking, colored petri nets, hashing, BPEL activities, the Spring framework, and the Java Architecture for XML Binding 2 API.
- In Chapter 5 we present two techniques for reducing the memory costs otherwise involved in model checking, which involve storing states as the difference from one of the neighboring states. Theoretical evaluation and experimental results indicate a significant reduction in memory requirements.

- In Chapter 6 we describe a novel method to reduce the time requirement for model checking a hierarchical model by exploring its inter dependent modules in parallel. These dependencies are stored as parameters in special data structures. On assigning specific values to these parameters, these dependencies are resolved and the reachability graph envisioned is obtained. The experimental results indicate a significant reduction in the time requirement.
- In Chapter 7 we introduce a technique to install hierarchy into a flat model by identifying the structural similarity. The technique is based on the decrease-and-conquer approach, wherein the bigger problem is broken into smaller problems and the solutions to the smaller problems are combined to solve the original problem. Compared to existing techniques, the model rendered is equivalent to the original model.
- In Chapter 8 we portray a verification framework to formalize a BPEL specification by transforming it into an XML-based formal model. This is done by extending the Spring framework and using JAXB 2 APIs. In addition, we determine a hierarchical relationship among BPEL activities to enhance the efficiency of this transformation. This framework (1) is extensible, (2) has a small amount of transformation time, and (3) can be used in combination with existing techniques.
- We conclude in Chapter 9 by summarizing the main achievements reported and listing possible directions for future research.

## REFERENCES

1. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, 2008.
2. 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*, May 2003.
3. J. A. Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of BPEL4WS business collaborations. In *E-Commerce and Web Technologies*, pages 76–85, 2004.
4. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, V. Mehta, S. Thatte, P. Yendluri, A. Yiu, and A. Alves. *Web Services Business Process Execution Language, Version 2.0*, December 2005.
5. A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '05, pages 1052–1059. IEEE Computer Society, Washington, DC, 2005.
6. G. Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, covers the 6th European Workshop on Applications and Theory in Petri Nets—Selected Papers*, pages 19–40. Springer-Verlag, Berlin, 1986.
7. F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. eflow: a platform for developing and managing composite e-services. In *Proceedings of the Academia/Industry*

*Working Conference on Research Challenges*, AIWORC '00, pages 341–348. IEEE Computer Society. Washington, DC, 2000.

8. D. Chakraborty, F. Perich, A. Joshi, T. W. Finin, and Y. Yesha. A reactive service composition architecture for pervasive computing environments. In *Proceedings of the IFIP TC6/WG6.8 Working Conference on Personal Wireless Communications*, PWC '02, pages 53–62, 2002.
9. P. P.-W. Chan and M. R. Lyu. Dynamic web service composition: a new approach in building reliable web service. In *Advanced Information Networking and Applications*, pages 20–25, 2008.
10. J. Chen and H. Cui. Translation from adapted UML to Promela for Corba-based applications. In *Proceedings of the 11th International SPIN Workshop*, pages 234–251, 2004.
11. S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, pages 450–464, 2001.
12. S. Christensen and L. Petrucci. Modular state space analysis of colored petri nets. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets ICATPN '95*, pages 201–217. Springer-Verlag, Berlin, 1995.
13. C. Z. Garrett. Software modeling introduction. *Borland White Paper*. Borland Software Corporation, Austin, TX, March 2003.
14. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, Berlin, 1998.
15. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.
16. W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly & Associates, Sebastopol, CA, 2003.
17. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language for Web Services, Version 1.0*. IBM, Microsoft, BEA Systems, July 2002.
18. L. Elgaard. *The Symmetry Method for Colored Petri Nets: Theory, Tools and Practical Use*. Ph.D. dissertation, University of Aarhus, Denmark, 2002.
19. E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1–2):105–131, 1996.
20. S. Evangelista, S. Haddad, and J.-F. Pradat-Peyre. Syntactical colored petri nets reductions. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis, ATVA 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 202–216. Springer-Verlag, Berlin, 2005.
21. S. Evangelista and J.-F. Pradat-Peyre. On the computation of stubborn sets of colored petri nets. In *Proceedings of the 27th International Conference on Application and Theory of Petri Nets, ICATPN '06*, pages 146–165. Springer-Verlag, Berlin, 2006.
22. S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, Berlin, 2005.

23. D. Fahland, W. Reisig, and D. Fahland. ASM-based semantics for BPEL: the negative control flow. In *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 131–151, 2005.
24. D. Fahland. Complete abstract operational semantics for the web service business process execution language. *Informatik-Berichte 190*. Humboldt-Universität zu Berlin, September 2005.
25. A. Ferrara. Web services: a process algebra approach. In *Proceedings of the 2nd International Conference on Service Oriented Computing, ICSOC '04*, pages 242–251, 2004.
26. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003*, pages 152–161, October 2003.
27. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th World Wide Web Conference*, pages 621–630. ACM, New York, 2004.
28. Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime monitoring of web service conversations. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 42–57. ACM, New York, 2007.
29. J. Geldenhuys and A. Valmari. A nearly memory-optimal data structure for sets and mappings. In *Proceedings of the 10th International SPIN Conference on Model Checking Software*, pages 136–150. Springer-Verlag, Berlin, 2003.
30. J.-C. Grégoire. State space compression in spin with GETSs. In *Proceedings of the 2nd SPIN Workshop*, pages 3–19. American Mathematical Society, Providence, RI, 1996.
31. O. Grumberg and H. Veith, Eds. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2008.
32. S. Haddad. A reduction theory for colored nets. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets—Selected Papers*, pages 209–235. Springer-Verlag, Berlin, 1990.
33. A. Hall. Realising the benefits of formal methods. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, pages 1–4, 2005.
34. G. J. Holzmann. State compression in spin: recursive indexing and compression training runs. In *Proceedings of the 3rd International SPIN Workshop*, 1997.
35. G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2:270–278, 1999.
36. J. Hu, C. Guo, H. Wang, and P. Zou. Web services peer-to-peer discovery service for automated web service composition. In *Proceedings of the 3rd International Conference on Network and Mobile Computing, ICCNMC '05*, pages 509–518, 2005.
37. F. Leyman. *Web Services Flow Language, Version 1.0*. IBM, Armonk, NY, May 2001.
38. D. Jacobs. Distributed computing with bea weblogic server. In *Proceedings of the 2003 CIDR Conference*, 2003.
39. K. Jensen and L. M. Kristensen. *Colored Petri Nets: Modelling and Validation of Concurrent Systems*. Springer-Verlag, Berlin, 2009.
40. H. Kang, X. Yang, and S. Yuan. Modeling and verification of web services composition based on cpn. In *Proceedings of the 2007 IFIP International Conference on Network and*

*Parallel Computing Workshops*, pages 613–617. IEEE Computer Society, Washington, DC, 2007.

41. L. M. Kristensen and A. Valmari. Finding stubborn sets of colored petri nets without unfolding. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets, ICATPN '98*; pages 104–123. Springer-Verlag, Berlin, 1998.
42. G. J. Myers. *Art of Software Testing*. J. Wiley, New York, 1979.
43. B. Parreaux. Difference compression in spin. In *Proceedings of the 4th SPIN Workshop*, 1998.
44. S. R. Ponnekanti and A. Fox. Sword: a developer toolkit for web service composition. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, HI, 2002.
45. J. Rushby. Formal methods and critical systems in the real world. In *Formal Methods for Trustworthy Computer Systems, FM '89*, pages 121–125, 1989.
46. K. Schmidt and C. Stahl. A petri net semantic for BPEL4WS validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets, AWPN '04*, pages 1–6, 2004.
47. K. Schmidt. Using petri net invariants in state space construction. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003*, pages 473–488. Springer-Verlag, Berlin, 2003.
48. J. C. Sloan and T. M. Khoshgoftaar. From web service artifact to a readable and verifiable model. *IEEE Transactions on Services Computing*, 2:277–288, October 2009.
49. C. Stahl. A petri net semantics for BPEL. *Informatik-Berichte 188*. Humboldt-Universität zu Berlin, July 2005.
50. S. Thatte. *XLANG Web Services for Business Process Design*. Microsoft Corporation, Redmond, CA, May 2001.
51. W. M. P. van der Aalst. Don't go with the flow: web services composition standards exposed. *IEEE Intelligent Systems*, January–February 2003.
52. C. F. Vasters. *Biztalk Server 2000: A Beginner's Guide*. McGraw-Hill, New York, 2001.
53. W. Visser. Memory efficient state storage in spin. In *Proceedings of the 2nd SPIN Workshop*, pages 21–35. American Mathematical Society, Providence, RI, 1996.
54. P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Pattern-based analysis of BPEL4WS. *QUT Technical Report FIT-TR-2002-04*. Queensland University of Technology, Brisbane, Australia, 2002.
55. Y.-P. Yang, Q.-P. Tan, Y. Xiao, J.-Shan Yu, and F. Liu. Verifying web services composition: a transformation-based approach. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 546–548. IEEE Computer Society, Washington, DC, 2005.
56. X. Yi and K. J. Kochut. A cp-nets-based design and verification framework for web services composition. In *Proceedings of the IEEE International Conference on Web Services, ICWS '04*; page 756. IEEE Computer Society, Washington, DC, USA, 2004.
57. L. Zeng. *Dynamic Web Services Composition*. Ph.D. dissertation, University of New South Wales, Australia, 2003.