Formal Verification Issues For Component-Based Development

Mehdi Hariati
Computer Science Department, LISCO Laboratory Badji Mokhtar-Annaba University, Annaba, Algeria
E-mail: mehdi.hariati@gmail.com

Keywords: component-based development, formal verification, classification.

Received: May 6, 2020.

Component-based development has made a breakthrough in software industry, it offers safer systems and easier to maintain, furthermore, costs and time to market are reduced. However, several issues, such as the correctness of component-based systems, their adaptation or the interactions between their components, require rigorous verification through the use of formal methods and tools. In this paper, we first present an introduction to component-based development; afterward we propose a classification of formal verification issues for component-based systems.

Povzetek: V tem članku je predstavljena klasifikacija formalnih metod preverjanja za sisteme, ki temeljijo na komponentah.

1 Introduction

In component-based development [1] the construction of a software system is reduced to an assembly of separately developed software components. This offers as advantages to reduce development costs as well as time to market. Moreover, the quality of the software systems is better, since the latter are built from tested and certified components. In addition, the maintenance and evolution stages of the system are simply a replacement of software components; furthermore, in response to changes in users' requirements or in the environment, component-based systems can also be reconfigured by modifying the links of their architecture.

Nevertheless, the component-based development process should be controlled by the use of formal methods, which allow, at any stage of the lifecycle, verifying important issues; such as the correctness of component-based systems, their adaptation or the interactions between their software components.

This paper is structured as follows. Section 2 presents the basic concepts of component-based development. In Section 3 we show the need for the use of formal methods through a classification of the various verification issues for component-based systems. Section 4 is devoted for related work . Section 5 presents a typical application domain, namely, Web Services. Finally, section 6 concludes this paper.

2 Related work

As to the best of our knowledge, this paper is the first presenting a classification of the main issues of formal verifications for the component-based systems, nevertheless, other works deal with the need for the formalization in this domain. In [30], the authors present the need for an abstract approach, the need for formalization for architecture description languages and interface description languages, and the formal languages used for formalization. Compared to our work, the authors invest much more in the study and comparison of the formal languages used in the field of software components, while our work rather focuses on the identification and classification of the problems that may arise during the component based development.

The authors of [31] present briefly an introduction to the component-based development; afterwards the need for formalization in this context is illustrated through a non-trivial example. However, the authors do not offer a detailed classification of potential problems of component-based development.

In [29], a classification of component models is proposed through a comparative study in five dimensions: life cycle, interface specification, interactions, extra-functional properties, and domains. Indeed, this work constitutes a more general classification of component models; the authors introduce the use of formal languages for software components, however, they do not provide a detailed study of formal verification issues for component-based development.

Further, unlike [30] and [31], in order to be more self-contained, basic concepts related to component-based development are provided, this is essential for understanding the formal verification issues.

3 Basic concepts of component-based development

In this section we present the basic principles and concepts of component-based development.

3.1 Software component

In the literature, there are many definitions of the notion of software component; according to [1], "A software

component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition". Indeed, a software component interacts with its environment only through its interfaces, since it is designed without any knowledge of its environment; this offers an independence allowing its use in different contexts.

3.2 Interfaces and assembly

A software component can have two types of interfaces: on the one hand, the provided interfaces; they represent the services that the component offers, on the other hand, the required interfaces; which are the services that the component needs to accomplish its functions. The assembly of a component-based system is done by linking the provided interfaces with the required interfaces of a selection of software components; however, in order to guarantee a correct assembly of these components, the compatibility of their interfaces should be verified beforehand.

The semantics of an interface is usually specified by its signature. However, the description of an interface only by its signature is insufficient for modeling and verifying the notion of compatibility, indeed, the specification of an interface must also include the definition of the behavior, such as the sequence of service calls between components of the system, or the time constraints, such as the execution time of a service. As we will see in the next sections, the application of formal methods is inescapable for the verification of these issues.

3.3 Component models and component frameworks

Others aspects, relating in particular to the definition of the components and their composition are specified by the component model to which the component is assigned. Indeed, the component models define a specific representation, composition modes, interaction styles and others standards dedicated to software components [2]. In addition, component models form the basis for creating *component frameworks*.

Component frameworks establish the physical environmental conditions for the execution and cooperation of components in the system, and they help also to regulate the interactions between components in execution [1].

Component frameworks can only concern physical components, unlike component models, these can be defined for the different levels of abstraction for a component [3], indeed, some component models define a software component as an execution entity, this is the case for Fractal [4] for example, while others component models define a software component as a design entity, as is the case for SOFA [5].

3.4 Instance of a software component

Some component models distinguish component types from their instances, allowing the creation and the destruction of component instances at runtime, as is the case for EJB [6] or CCM [7]. Others component models like Wright [8] do not take instantiation into account.

3.5 Synchronous communication vs asynchronous communication

Usually, the communication between the software components is done in a synchronous manner, as is the case for Darwin [9] and SOFA. However, in some models such as EJB or CCM, communication can be done by asynchronously sending and receiving messages.

3.6 Flat models vs hierarchical models

A set of basic software components can be assembled to give a composite component. In flat component models, this composite component represents the final component-based system, as is the case for EJB or CCM. However, in hierarchical component models, such as SOFA or Fractal, the composite component may in turn be subject to composition with others components, allowing the construction of a component-based system with several hierarchical levels of components. Furthermore, in hierarchical models, we must specify the interfaces to be delegated outside a composite component to be linked to compatible interfaces in the higher hierarchical levels of composition.

3.7 Single binding vs multiple binding

Some component models suppose one-to-one linking of interfaces, i.e. single bindings, as in SOFA, others component models allow an interface to be linked to several others interfaces, i.e. multiple bindings, as is the case of EJB and Fractal.

3.8 Life cycle of a component-based system

Component-based software systems are developed by selecting and assembling *off-the-shelf components*, instead of being programmed, this makes the lifecycle of a component-based software system different from traditional software system; it mainly comprises the following steps:

1- *Requirements specification*: It concerns collecting, analyzing and specifying the needs of the future users of the system.
2- *Architecture specification*: The architecture of the software specifies the system in terms of abstract components of design and interactions between these components.
3- *Selection and customization of components*: First, the concrete components taken on the shelf are selected according to the software architecture; in a second step, each component must be

personalized before being integrated into the new system.

4- *Integration of the system*: Integration is achieved by establishing mechanisms for communication and coordination of the various components of the final software system.
5- *Test of the system*: Various methods and tools are used to test the component-based system; in fact, it is a question of checking the properties concerning functional aspects as well as those related to the quality of the software.
6- *Deployment*: This is the installation of the software components of the system on one or more computers.
7- *Maintenance and evolution of the system*: After deployment, parts of the component-based system can be modified, due to changes in users' requirements or in the environment.

The concept of software construction by reuse is not new, indeed, the idea was already present in object-oriented programming, it was implemented by the inheritance mechanism; the relatively recent emergence of new technologies has significantly increased the possibilities of building systems and applications from reusable components. Furthermore, building systems based on components or building components for systems in different application areas requires methodologies and processes, including not only development and maintenance aspects, but also those relating to organizational, marketing, legal and other aspects.

3.9 Development for reuse and development through reuse

The component-based software engineering process includes two separate but linked processes via a component market. In the following we present each of the two processes:

- *Development for Reuse*: This process consists of an analysis of the application domains in order to develop commercial-off-the-shelf (COTS) components related to these domains. To complete a successful reuse of the software, standards for similar systems must be identified and represented in a form that can be easily exploited to build other systems in the domain. Once created, reusable components will be available in organizations or at the market level as commercial components.
- *Development through reuse*: this is related to the assembly of software systems from the components taken on the shelf.

3.10 The objectives of component-based development

The main objectives of component-based development can be summarized as follows:

- *Reuse*: This is the main objective of component-based development. While some software components of a large system are necessarily special purpose components, it is imperative to design and assemble components in order to reuse them in the development of others systems.
- *Independent development of software components*: Large software systems should be able to be assembled from components developed by different people, for this purpose, it is essential to decouple the developers from the components of their users, this is done mainly through the specifications of the behavior of components.
- *Software quality*: A software component or a component-based system should have the desired behavior. Quality assurance technologies for component-based software systems are currently relatively premature, as the characteristics of component-based systems differ from those of conventional systems.
- *Maintainability*: A component-based system should be built in a way that is understandable and easy to evolve.

3.11 The contributions of component-based development

The contributions of component-based development can be presented as follows:

- *More efficient management of complexity*: The division of large and complex systems into sub-systems offers greater control over their complexity.
- *Time to market is reduced*: Component-based development consists of assembling existing components, which reduces development time, and therefore accelerates the time to market.
- *Costs are reduced*: While some software components are completely specific to a given application, other software components can be reused and shared with other developers, thereby reducing their costs by damping through a large population.
- *Quality is improved*: Component-based development greatly improves the quality of the systems, since the latter are built from components that are already tested and certified.
- *Easier maintenance and evolution*: The maintenance and evolution of component-based systems is easier, since most of the time they are

reduced to simple additions, deletions or replacement of software components.

4 Classification of formal verification issues for component-based systems

Formal approaches are rigorous methods aimed at modeling and analyzing complex systems. The idea of verifying programs is not new; in fact it dates back to the 1960s. Today, formal techniques and tools are widely used in both the academic and the industrial worlds.

In our context, formal methods are essential for component-based development because they enable addressing important verification issues throughout the lifecycle of a component-based system. In the remainder of this section, we will detail these verification issues which we have classified into three levels, namely, at an individual component, during the composition of the components, and finally at the evolution level.

4.1 Component level

This level of analysis addresses the verification of an individual component before its composition with the rest of the system; we classified this verification into two types:

- Context-independent verification: it consists of verifying the properties of a component in the isolation, thereby independently of its deployment context; indeed, the issues to be checked can concern the absence of deadlock in its own specification or the coherence of the specification of its temporal constraints.
- Context-dependent verification: In component-based development, components are developed independently of their deployment context; therefore, component correctness can be very difficult to define, as a component may behave correctly in a context but incorrectly in another. Existing approaches remedy this situation in two different ways; some approaches [10, 11] propose to attribute to each component a description of its properties, thereby enabling the user of component to decide if the latter can behave correctly in a given context. Other approaches [12, 13] deliver software components with a set of quality properties that are guaranteed in all contexts satisfying a number of conditions.

4.2 Composition level

This level addresses the verification of the composition of the system; we classified this verification into tree main issues:

4.2.1 Compatibility of components

The software components constituting a component-based system can be delivered by different sellers; therefore verification of their compatibility is an important issue. Some approaches define compatibility only in terms of signatures of services linking components [14, 7, 15]. However, this description is by no means exhaustive, because it does not include for example, the specification of the services calls sequence of a component, such an aspect is more a matter of behavior. On the other hand, other approaches offer a richer description of compatibility, including description of the behavior [16]. This makes it possible to verify that the composition will not lead to an erroneous interaction between the components of the system.

Some approaches propose to verify compatibility at design time, while others perform checks during execution, thereby detect bad interactions between components dynamically; using a test environment in which the concerned components are duplicated [17].

Moreover, even if the components are not completely incompatible, they can sometimes cooperate correctly by generating appropriate adapters of their interfaces. Some approaches generate adapters for connecting components belonging to different component models [18, 29]; this can be done in a fully automatic manner. Other approaches include adapters for integrating an incompatible functionality of components [19], in which case additional input is required from the user or the monitoring phase to provide information concerning the parts corresponding to the incompatible functionality.

4.2.2 Assembly of components

The process of assembling components is mainly twofold: identifying the correct components taken on the shelf, and their connections together, so that the resulting component-based system corresponds to the desired requirements.

Usually, assembly strategies focus on finding the most cost-effective solution with respect to time [19]. The cost function can, for example, evaluate the components in terms of their performance measurements or the minimization of new requirements generated by the added components. The assembly can be selected based on an exhaustive evaluation of all possible alternatives [20], or via an iterative construction of a relatively optimal solution [21].

In this context, formal methods make the problem of assembly of components considerably simpler by simply providing a design of the component based system comprising specifications of a set of components and their connections, the problem being reduced to simply finding the correct component implementations taken on the shelf and formally verifying their compliance with the expected specifications.

4.2.3 The global verification

Formal methods are very useful for verifying the global properties of a final component-based system. In this case, formal analysis generally includes:

- Verification of standard coordination errors.

- The absence of deadlock in the system.
- Verification of the different time constraints in the global system.
- The order of execution of a set of services of a components selection in the final system.
- Verification of the number of components that can simultaneously access to the same service.

This verification can be carried out on the whole of the final component-based system or simply on a well-defined part.

Furthermore, in addition to checking properties, formal methods can also help in optimizing component-based systems, namely:

- Detection of inactive components, which can be removed from the system.
- The search for optimal system deployment by placing components in compute nodes based on the density of interaction between them [22].

As with compatibility, some approaches check the properties of a global system at design time, while other approaches allow dynamic verification of the system, in fact, the conformance of the current behavior of the components in execution is verified in parallel with its specification [27], thereby any errors are reported in case of discrepancy.

4.3 Evolution level

After the deployment phase, a component-based system can evolve or adapt, in response to changes in users' needs or changes in its environment [23], namely: interoperability with others systems, optimization of computational algorithms, or technical changes.

Formal methods and techniques are very useful for modeling and analyzing the evolution of component-based systems [24]. We have classified this analysis into two types:

- *The dynamic reconfiguration of the architecture*: this mainly includes the change of the links between the system components as well as the creation and destruction of the instances of the components. At this level, formal analysis seeks to verify the coherence of the global system after a dynamic reconfiguration.
- *Substitutability*: one or more components can be replaced with new ones. Generally, approaches addressing this issue define an equivalence relation between the old and the new component, in order to verify that the substitution does not violate the correctness of the global system [25]. However, in some cases, the verification of the equivalence between the two versions of the system is not necessarily strong, because it is only necessary that the new system satisfies a given explicit property, this is considered much more by the approaches that do not aim to guarantee that the behavior remains unchanged, but rather to identify the behavioral differences between several versions of the system [26].

Furthermore, the evolution of a component-based system is usually defined with a set of evolution rules.

5 An application domain: Web services

Web services are a typical application domain of component-based development. Indeed, formal methods, used pragmatically, represent a very powerful way to verify several issues, such as the description, composition or evolution of web services.

Regarding the verification of the composition, for instance, the goal is to find the best way to put the services together for the accomplishment of a global task. The composition of web services is called choreography. Nowadays, several languages are dedicated to the description of choreography, for example: WS-CDL (Web Services Choreography Description Language) [32] or WSCI (Web Service Choreography Interface) [33].

Another example of the formal verification for web services is orchestration, this describes the business logic of web services; in fact, it is the description of the control flow of business processes, such as: sequential or parallel execution, etc. WS-BPEL (Web Services Business Process Execution Language) [34] is one of the most widely used languages to describe orchestration.

In this context, formal verification tools perform translations from languages such as: WS-CDL or WS-BPEL, to formalisms, such as: process algebras [8] or timed automata [35], thus allowing the verification of requested properties.

6 Conclusion

We presented an overview of the principles and basic concepts of the component-based software development paradigm. Afterwards, through a classification of verification issues for software components, we have shown the need for formal methods and techniques in this context. More generally, for a real integration of formal methods into the component-based development process, frameworks with textual input languages or graphical notations must be provided, and translation algorithms must be implemented; including translations between informal concepts of component-based systems to formalisms, as well as translations of these formalisms to proof or verification tools such as model checking tools.

Further, other issues have yet to be solved. In fact, we have good techniques and tools for formal verifications dedicated to the design phase, such as the UPPAAL model checker [28]; however, these tools cannot be used to do verifications during the execution phase, to control the

behavior of a running system with respect to an expected formal model. On the other hand, it would be practical to design tools that allow direct generation of code from the formal specification of a component-based system.

7 Acknowledgement

The authors would like to thank the DGRSDT (General Directorate of Scientific Research and Technological Development) - MESRS (Ministry of Higher Education and Scientific Research), ALGERIA, for the financial support of LISCO Laboratory.

8 References

[1] C. Szyperski. Component Software Beyond Object-Oriented Programming. Addison-Wesley, USA, 2. edition, 2002. ISBN 0-201-74572-0.

[2] G. T. Heineman and W. T. Councill. Component Based Software Engineering - Putting the Pieces Together. Addison-Wesley, USA, May 2001. ISBN 0-201-70485-4.

[3] A. Rausch, R. Reussner, and al, editors. The Common Component Modeling Example: Comparing Software Component Models. To appear in LNCS, 2008.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal Component Model and its Support in Java. Software: Practice and Experience, 36(11- 12): 1257-1284, August 2006.

[5] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. IEEE Transactions on Software Engineering, 28(11): 1056-1076, November 2002.

[6] Sun Microsystems. Enterprise JavaBeans 3.0 Specification, May 2006.

[7] Object Management Group. CORBA Component Model 4.0 Specification. Technical Report formal/06-04-01, Object Management Group, April 2006.

[8] R. J. Allen. A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon University, School of Computer Science, USA, May 1997.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In Proceedings of the 5th European Software Engineering Conference (ESEC'95), volume 989 of LNCS, pages: 137- 153. Springer-Verlag, September 1995.

[10] B. Meyer. The Grand Challenge of Trusted Components. In Proceedings of the 25th International Conference on Software Engineering (ICSE'03), pages: 660-667. IEEE Computer Society, May 2003.

[11] B. Meyer, C. Mingins, and H. Schmidt. Providing Trusted Components to the Industry. Computer, 31(5): 104-105, May 1998.

[12] G. Xie. Decompositional Verification of Component-based Systems - A Hybrid Approach. In Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'04), pages 414-417. IEEE Computer Society, September 2004.

[13] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning Assumptions for Compositional Verification. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), volume 2619 of LNCS, pages: 331-346. Springer-Verlag, January 2003.

[14] M. Corporation. COM: Component Object Model Technologies, December 2007. URL http://www.microsoft.com/com/.

[15] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. CWI Quarterly, 2(3): 219-246, September 1989.

[16] L. de Alfaro and T. A. Henzinger. Interface-based Design. In Proceedings of the 2004 Marktoberdorf Summer School, pages 1-25. Kluwer, The Netherlands, 2005.

[17] D. Niebuhr and A. Rausch. A concept for dynamic wiring of components: correctness in dynamic adaptive systems. In Proceedings of the ESEC/FSE Conference on Specification and Verification of Component-Based Systems (SAVCBS'07), pages 101-102. ACM Press, September 2007.

[18] O. Galk and T. Bures. Generating Connectors for Heterogeneous Deployment. In Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM'05), pages 54-61. ACM Press, September 2005.

[19] L. Gesellensetter and S. Glesner. Only the Best Can Make It: Optimal Component Selection. Electronic Notes in Theoretical Computer Science (ENTCS), 176(2): 105-124, May 2007.

[20] N. Barthwal and M. Woodside. E±cient Evaluation of Alternatives for Assembly of Services. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'05), pages 1-8. IEEE Computer Society, April 2005.

[21] N. Desnos, S. Vauttier, C. Urtado, and M. Huchard. Software Architecture, volume 4344 of LNCS, chapter Automating the Building of Software Component Architectures, pages 228-235. Springer-Verlag, December 2006.

[22] B. Zimmerova. Component Placement in Distributed Environment w.r.t. Component Interaction. In Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), pages: 260-267. FIT VUT Brno,Czech Republic, October 2006.

[23] P. Waewsawangwong. A Constraint Architectural Description Approach to Self-Organising Component-Based Software Systems. In Proceedings of the International Conference on Software Engineering (ICSE'04), pages: 81-83. IEEE Computer Society,May 2004.

[24] B. Zimmerova and P. Varekova. Reecting Creation and Destruction of Instances in CBSs Modelling and

Verification. In Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'07), pages: 257-264. Novotny, Brno, Czech Republic, October 2007.

[25] P. Parzek, F. Plasil, and J. Kofro·n. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In Proceedings of the Software Engineering Workshop (SEW'06), pages: 133-141. IEEE Computer Society,April 2006.

[26] L. Mariani and M. Pezzµe. A Technique for Verifying Component-Based Software. In Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS'04), volume 116 of ENTCS, pages: 17-30. Elsevier Science Publishers,January 2005.

[27] P. Parzek, F. Plasil, and J. Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In Proceedings of the Software Engineering Workshop (SEW'06), pages: 133-141. IEEE Computer Society,April 2006.

[28] LARSEN, Kim G., PETTERSSON, Paul, et YI, Wang. UPPAAL in a nutshell. International journal on software tools for technology transfer, 1997, vol. 1, no 1-2, p. 134-152.

[29] CRNKOVIC, Ivica, CHAUDRON, Michel, SENTILLES, Séverine, et al. A classification framework for component models. Software Engineering Research and Practice in Sweden, 2007, p. 3.

[30] POIZAT, Pascal, ROYER, Jean-Claude, et SALAÜN, Gwen. Formal methods for component description, coordination and adaptation. Canal et al.[4], 2004, p. 89-100.

[31] MAKOWSKI, Piotr et RAVN, Anders P. Component Based Development-Where is the Place for Formalization?. 2003.

[32] KAVANTZAS, Nickolas, BURDETT, David, RITZINGER, Gregory, et al. Web service choreography description language (wscdl) 1.0. 2004.

[33] ARKIN, Assaf, ASKARY, Sid, FORDIN, Scott, et al. Web service choreography interface (WSCI) 1.0, 2002. URL http://www. w3. org/TR/wsci, 2002.

[34] ARKIN, Assaf, ASKARY, Sid, BLOCH, Ben, et al. Web services business process execution language version 2.0. Working Draft. WS-BPEL TC OASIS, 2005.

[35] ALUR, Rajeev et DILL, David L. A theory of timed automata. Theoretical computer science, 1994, vol. 126, no 2, p. 183-235.