Model Checking for Software Architectures*

Radu Mateescu

Inria Rhône-Alpes / Vasy, 655, avenue de l'Europe
F-38330 Montbonnot Saint Martin, France
Radu.Mateescu@inria.fr

Abstract. Software architectures are engineering artifacts which provide high-level descriptions of complex systems. Certain recent architecture description languages (Adls) allow to represent a system's structure and behaviour together with its dynamic changes and evolutions. Model checking techniques offer a useful way for automatically verifying finite-state Adl descriptions w.r.t. their desired correctness requirements. In this position paper, we discuss several issues related to the application of model checking in the area of software architectures, underlining the aspects of interest for current and future research (construction of state spaces, expression and verification of requirements, state explosion).

1 Introduction

Software architectures [27] are essential engineering artifacts used in the design process of complex software systems. They specify at a high abstraction level various aspects of a system: gross organization into components, protocols for communication and data access, functionality of design elements, etc. Over the last decade, a significant number of architecture description languages (Adls) were proposed and supported by dedicated tool environments (see [20] for a survey). Recently defined Adls such as π-Space [5] and the ArchWare Adl [22] aim at describing the structure and behaviour of software systems that are subject to dynamic changes and evolutions. Inspired from mobile process calculi, such as the higher-order polyadic π-calculus [21], these Adls provide mobility of communication channels, dynamic process creation/destruction, and higher-order process handling, enabling one to design evolvable, industrial-scale systems. To ensure the reliability of such complex systems, computer-assisted verification methodologies become a necessary step in the design process.

Model checking [6] is a verification technique well-adapted for the automatic detection of errors in complex systems. Starting from a formal representation of the system under design, e.g., an Adl description, a corresponding model (state space) is constructed; then, the desired correctness requirements, expressed in temporal logic, are checked on the resulting model by using specific algorithms. Although limited to finite-state systems, model checking provides a simple and efficient verification approach, particularly useful in the early phases of the design process, when errors are likely to occur more frequently.

* This study was supported by the European Ist-2001-32360 project "ArchWare".

During the last decade, model checking was successfully applied for analysing software architectures described using different ADLs inspired from process algebras, the most prominent ones being WRIGHT [2] and DARWIN [17]. WRIGHT is based upon CSP, thus allowing to use FDR [25] to perform various architectural consistency checks amenable to deadlock detection and behavioural refinement. DARWIN uses the π-calculus to describe the structural aspects of the architecture (configuration and coordination) and FSP (a dialect of CSP) to describe the behaviour of individual components; this allows to check properties expressed in Linear Temporal Logic (LTL) using LTSA [16].

However, so far relatively little work was dedicated to model checking for ADLs that provide mobility and dynamicity mechanisms. Dynamic WRIGHT [3] allows to describe dynamic reconfiguration and steady-state (static) behaviour orthogonally, by introducing special reconfiguration-triggering events handled by a configuror process; reconfigurable architectures are translated into CSP by instantiating a finite number of possible configurations and by tagging events with the configuration in which they occur. A similar approach was used for ΔPADL [1], where dynamic architectures are simulated by instantiating finite numbers of replicas and by adding transparent routers to model dynamic reconfiguration. In addition to these general results concerning the extension of ADLs with dynamicity, the problem of model checking was also considered for dynamic systems belonging to specific domains, such as publish-subscribe systems [12].

In this position paper, we discuss three different aspects related to the application of model checking techniques for analysing dynamic ADL descriptions: construction of the state space corresponding to an ADL description (Section 2), expression and verification of correctness requirements (Section 3), and handling of the state explosion problem (Section 4). Finally, we give some concluding remarks and directions for future research (Section 5).

2 Constructing State Spaces

We can identify two ways of building the state space of an architectural description written in a dynamic ADL: either by developing from scratch an ADL simulator able to explore all reachable states of an architectural description, or by translating the ADL into another formal specification language already equipped with a state space generator. The first solution would certainly be the most efficient and accurate w.r.t. the operational semantics of the ADL, but may require a considerable effort (e.g., simulators for the polyadic π-calculus, such as MWB [29], are complex pieces of software). On the other hand, the second solution can be much simpler to achieve and may take advantage of the software tools already available for the target language. In the sequel, we examine the latter solution by considering as targets LOTOS [14] and E-LOTOS [15], two languages standardized by ISO, which combine the best features of classical value-passing process algebras (CCS and CSP) and are equipped with state-of-the-art software engineering environments such as the CADP verification toolbox [9].

Dynamic process creation. To obtain finite-state ADL descriptions in presence of dynamic process creation, one must statically bound the maximum number of process replicas that may coexist. LOTOS can describe dynamic process creation by using recursion through parallel composition (e.g., processes like `P := a; stop ||| P`), but most of the existing compilers do not handle this construct, since it may yield infinite, non regular behaviours. A solution would consist in statically expanding each dynamic process into the parallel composition of its n allowed replicas (all initially idle): this can be expressed concisely in E-LOTOS by using the indexed parallel composition operator [11]. Alternatively, one may directly construct the sequential process equivalent to the interleaving of n parallel replicas [13].

Mobility of communication channels. LOTOS and E-LOTOS assume that the process network has a fixed communication topology. Nevertheless, mobility of communication channels can be simulated in LOTOS by defining a data type "channel name", which allows to send and receive channel names as ordinary data values. The LOTOS processes produced by translating ADL (dynamic) processes will still communicate along a fixed network of gates, but each communication on a gate `G` will carry the name of an underlying mobile channel (e.g., `G !c !0` denotes the emission of value 0 along channel `c`). The number of gates can be reduced due to the powerful synchronization mechanisms of LOTOS, which allow several channels to be multiplexed on a single gate. Also, the fact of bounding the number of replicas for dynamic processes also induces a bound on the set of (private) channel names that can be created by individual processes.

Higher-order process handling. Since LOTOS provides only first-order constructs (data values and behaviours are clearly separated), it does not allow a direct representation of higher-order mechanisms such as sending a process along a channel. However, a significant part of a higher-order dynamic ADL can be translated into first-order by applying the translation from higher-order to first-order π-calculus [26, chap. 13].

By developing a translation according to the guidelines above, and by subsequently using a compiler for LOTOS (such as the CÆSAR [10] compiler of CADP), one can obtain a state space generator for a dynamic higher-order ADL. Such a tool would allow to handle finite-state configurations of ADL descriptions presenting a limited amount of dynamic process creation, channel mobility, and higher-order communication.

3 Checking Correctness Requirements

Temporal logics and μ-calculi [28] are well-studied formalisms for expressing correctness requirements of concurrent systems. During the last decade, many algorithms and model checking tools dedicated to these formalisms were developed; now, the research focuses on the application of these results in industrial context. We can mention two areas of interest w.r.t. the analysis of software architectures using temporal logics:

Optimized verification algorithms. The speed and memory performance of verification algorithms can still be improved, namely when they are applied to particular forms of models. For instance, *run-time verification* consists in analysing the behaviour of a system by checking correctness requirements on execution traces generated by executing or randomly simulating the system. In this context, memory-efficient verification algorithms have been designed for μ-calculus [18]; further improvements (e.g., memory consumption independent from the length of the trace) can be obtained by specialising these algorithms for particular temporal logics.

Advanced user interfaces. User-friendliness is essential for achieving an industrial usage of temporal logic. Several aspects must be considered when integrating model checking functionalities into an engineering environment: extension of the basic temporal logics with higher-level constructs, e.g., regular expressions [19]; identification of the interesting classes of requirements, which should be provided to the end-user by means of graphical and/or natural language interfaces; and automated interpretation of the diagnostics produced by model checkers in terms of the application under analysis.

4 Handling Large Systems

When using model checking to analyse large systems containing many parallel processes and complex data types – such as ADL descriptions of industrial systems – the size of the state space may become prohibitive, exceeding the available computing resources (the so-called *state explosion* problem). Several techniques were proposed for fighting against state explosion:

On-the-fly verification. Instead of constructing the state space entirely before checking correctness requirements (which may fail because of memory shortage), on-the-fly verification explores the state space incrementally, in a demand-driven way; this allows to detect errors in complex systems without constructing their whole state space explicitly. An open platform for developing generic on-the-fly verification tools is provided by the OPEN/CÆSAR environment [7] of CADP, together with various on-the-fly verification tools (guided simulation, searching of execution sequences, model checking, etc.).

Partial order reduction. Due to presence of independent components which evolve in parallel and do not synchronize directly, the state space of a parallel system often contains redundant interleavings of actions, which can be eliminated by applying partial order reductions [24]. A form of partial order reduction useful in the context of process algebras is τ-confluence, for which several tools are already available [23].

Compositional verification. Another way to avoid the explicit construction of the state space is by using abstraction and equivalence. Compositional verification consists in building the state spaces of the individual system's components, hiding the irrelevant actions (which denote internal activity), minimising the resulting state spaces according to an appropriate equivalence relation, and recomposing them in order to obtain the state space of the whole system. The SVL environment [8] of CADP provides an efficient and versatile framework for describing compositional verification scenarios.

Sufficient locality conditions. For specific correctness requirements (e.g., deadlock freedom), there exist sufficient conditions (e.g., acyclic interconnection topology) ensuring the satisfaction of the requirement on the whole system by checking it locally on each component of the system [4]. In this way state explosion is avoided, since only the state spaces of the individual components need to be constructed. An interesting issue concerns the extension of these results for more elaborate correctness requirements.

Experience has shown that analysis of large systems can be achieved effectively by combining different methods. A promising direction of research would be to study the combination of the aforementioned verification methods in the field of software architectures, and to assess the results on real-life industrial systems.

5 Conclusion

In this position paper we have attempted to make precise several directions of research concerning the integration of model checking features within the design process of industrial systems based on software architectures and dynamic ADLS. At the present time, the theoretical developments underlying model checking have become mature, and robust, state-of-the-art tool environments are available. Therefore, we believe that a natural and effective way to proceed is by reusing, adapting, and enhancing the existing model checking technologies in the framework of software architectures.

References

1. P. Abate and M. Bernardo. A Scalable Approach to the Design of SW Architectures with Dynamically Created/Destroyed Components. In *Proc. of SEKE'02*, pp. 255–262, ACM, July 2002.
2. R. J. Allen. A Formal Approach to Software Architecture. Ph.D. Thesis, Technical Report CMU-CS-97-144, Carnegie Mellon University, May 1997.
3. R. J. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, LNCS vol. 1382, pp. 21–37.
4. M. Bernardo, P. Ciancarini, and L. Donatiello. Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems. In *Proc. of WICSA'01*, pp. 77–86. IEEE Computer Society, August 2001.
5. C. Chaudet and F. Oquendo. Pi-SPACE: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems. In *Proc. of ASE'2000*, pp. 245–248, September 2000.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
7. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, LNCS vol. 1384, pp. 68–84.
8. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *Proc. of FORTE'2001*, pp. 377–392. IFIP, Kluwer Academic Publishers, August 2001.

9. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, August 2002.
10. H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In *Proc. of PSTV'90* pp. 379–394, IFIP, June 1990.
11. H. Garavel and M. Sighireanu. Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS. In *Proc. of FMICS'98*, pp. 187–230. CWI, May 1998.
12. D. Garlan, S. Khersonsky, and J. S. Kim. Model Checking Publish-Subscribe Systems. In Proc. of SPIN'03, LNCS vol. 2648, pp. 166–180.
13. J. F. Groote. A Note on n Similar Parallel Processes. In *Proc. of FMICS'97*, pp. 65–75. CNR, July 1997.
14. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO Standard 8807, 1989.
15. ISO/IEC. Enhancements to LOTOS (E-LOTOS). ISO Standard 15437:2001.
16. J. Kramer, J. Magee, and S. Uchitel. Software Architecture Modeling & Analysis: A Rigorous Approach. In *Proc. of SFM'2003*, LNCS vol. 2804, pp. 44–51.
17. J. Magee, N. Dulay, S. Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In *Proc. of ESEC'95*, LNCS vol. 989, pp. 137–153.
18. R. Mateescu. Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems. In *Proc. of TACAS'2002*, LNCS vol. 2280, pp. 281–295.
19. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Sci. of Comp. Prog.*, 46(3):255–281, March 2003.
20. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
21. R. Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.
22. F. Oquendo, I. Alloui, S. Cîmpan, and H. Verjus. The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics. Project Deliverable D1.1b, European project IST 2001-32360 "ArchWare", December 2002.
23. G. Pace, F. Lang, and R. Mateescu. Calculating τ-Confluence Compositionally. In *Proc. of CAV'2003*, LNCS vol. 2725, pp. 446–459.
24. D. A. Peled, V. R. Pratt, and G. J. Holzmann, editors. *Partial Order Methods in Verification*, vol. 29 of DIMACS series. American Mathematical Society, 1997.
25. A.W. Roscoe. Model-Checking CSP. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994.
26. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
27. M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.
28. C. Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, 2001.
29. B. Victor and F. Moller. The Mobility Workbench – A Tool for the π-Calculus. In *Proc. of CAV'94*, LNCS vol. 818, pp. 428–440.