Performance Evaluation of Software Architectural Types: A Process Algebraic Approach

Marco Bernardo[1], Paolo Ciancarini[2], Lorenzo Donatiello[2]

[1]Università di Torino, Dipartimento di Informatica
Corso Svizzera 185, 10149 Torino, Italy
E-mail: bernardo@di.unito.it
[2]Università di Bologna, Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: { cianca, donat }@cs.unibo.it

The software architecture level of design allows to cope with the increasing size and complexity of software systems during the early stage of their development [7, 8]. To achieve this, the focus is turned from algorithmic and data structure related issues to the overall architecture of the system. The architecture is meant to be a collection of computational components together with a description of their connectors, i.e. the interactions between these components.

As software architecture emerges as a discipline within software engineering, it becomes increasingly important to support architectural development with languages and tools. It is widely recognized that suitable architectural description languages (ADLs for short) should be devised to formalize software architectures instead of using informal box-and-line diagrams, and related tools should be implemented to support the automatic analysis of architectural properties in order to allow the designer to make principled choices.

As far as we know, almost all the existing languages and tools deal only with functional aspects of software architectures. However, designers are often faced with the problem of choosing among different software architectures

which are functionally equivalent. This choice is thus driven by nonfunctional factors, and mostly by performance requirements. In general, as recognized in [9], performance analysis should be integrated into the software development process, starting from the earliest stages and continuing throughout the whole life cycle.

In order to create a framework where the functional and performance properties of formally represented software systems can be automatically evaluated at the architectural level, we need a suitable theory which provides the necessary underpinnings to the architectural concepts of component and connector. This is witnessed e.g. by the fact that a well known ADL like WRIGHT [1] is based on CSP [5].

In the field of computer and communication system modeling and analysis, several formal description techniques have been proposed in the last two decades which account for both functional and nonfunctional aspects of systems. Among such formal description techniques, there are stochastically timed process algebras (see, e.g., [2]). Their key feature is compositionality. First of all, like classical process algebras, they allow for compositional model construction because they are algebraic languages endowed with a small set of powerful operators, such as parallel composition, sequential composition, and alternative composition, which allow descriptions to be systematically built from their components. Unlike classical process algebras, stochastically timed process algebras come equipped with the capability of expressing activity durations by means of exponentially distributed random variables, so that the underlying performance models turn out to be Markov chains which can thus be exploited to effectively derive performance measures. Second, stochastically timed process algebras allow for compositional model manipulation. This is achieved by means of equivalences which relate terms possessing the same functional and performance properties. Whenever such equivalences are congruences, i.e. they are substitutive w.r.t. the algebraic operators, they permit to replace algebraic components with equivalent (smaller) ones without altering the overall system properties. Third, stochastically timed process algebras allow for compositional model solution whenever the underlying Markov chain meets certain conditions.

Since the compositionality of stochastically timed process algebras seems to be well suited for the architectural level of design, we propose their adoption for the development of an ADL aiming at predicting the performance of software systems and comparing the performance of several software ar-

chitectures designed for the same system. Such an ADL has been called ÆMPA [3] as it is based on the process algebra EMPA [2].

A description in ÆMPA is an architectural type:

archi_type ⟨name and parameters⟩
  archi_cc_types ⟨architectural component/connector types⟩
  archi_topology ⟨instances and attachments⟩
  archi_interactions ⟨architectural interactions⟩
end

Each architectural type is defined as a function of its component and connector types, its topology, its interactions, and its generic parameters. A component/connector type is in turn defined as a function of its behavior, specified as a family of EMPA sequential terms, and its interactions, specified as a set of EMPA action types. The architectural topology consists of a set of component/connector instances related by a set of attachments among their interactions. Architectural interactions are interactions of component instances which support hierarchical architectural modeling. Finally, generic parameters are basically values for parametric rates and weights.

For the sake of ease of use, the textual notation above is accompanied by a graphical notation inspired by the flow graphs of [6], as they may provide a visual help to the development of architectural descriptions. A flow graph is a network of nodes each equipped with a set of ports; two ports of two different nodes are linked together if the two nodes can interact. Given an ÆMPA description, each component/connector type can be represented as a node (depicted as a rounded box) with its behavior textually reported inside the node and its interactions labeling the ports (depicted as black circles, or white squares in the case they refer to the whole architecture). We then create an instance of a node for each instance of the component/connector type it refers to. Finally, nodes are linked together according to the specified attachments. In case of hierarchical architectural modeling, a node can contain a flow graph whose architectural interactions are linked (through dashed lines) to the ports of the node.

The concept of architectural type, proposed in [4], is an abstraction of the concept of architectural style. Given an architectural style, the set of component and connector instances and their internal behavior can vary from architectural instance to architectural instance, but the structure of the overall interconnection of component and connector instances and their internal behavior w.r.t. interactions is fixed. Because of the presence of two degrees

of freedom (variability of the set of component and connector instances and variability of their internal behavior), investigating the properties which are common to all the instances of an architectural style is not an easy task. To make such a task manageable, architectural types are advocated in [4] because they constrain the set of component and connector instances to be fixed. The instances of a given architectural type are then generated by letting the behavior of component and connector types vary. In other words, the component/connector types specified in an architectural type are viewed as being formal, so one can call for an architectural type and pass to it actual component/connector types.

The formal semantics for ÆMPA is given by translation into EMPA by essentially exploiting the parallel composition operator. Given that the semantics of a component/connector type is the family of EMPA sequential terms expressing its behavior and that the semantics of a component/connector instance is the semantics of the related type, the semantics of an architectural type is obtained by composing in parallel the semantics of the component/connector instances according to the specified attachments. Once the corresponding family of EMPA terms has been generated, the analysis of the architectural properties can be carried out by means of existing tools. In particular, it is possible to verify functional properties (like deadlock freedom and mutually exclusive use of resources) through model checking, equivalence checking, and preorder checking. Likewise, it is possible to evaluate performance measures (such as system throughput and user response time) via exact or approximate Markovian analysis or simulation.

We observe that, from the process algebra perspective, creating an ADL can be viewed as an attempt to force the designer to model systems in a more controlled way, which in particular elucidates the basic architectural concepts of component and connector and hopefully enhances the usability of process algebras. However, this syntactic sugar alone is not enough to create a useful ADL. It must be accompanied by suitable techniques to verify the well formedness of architectural descriptions, such as the architectural compatibility and conformity checking. The purpose of the former check is to ensure that an architectural type is well connected, in the sense that every pair composed of a component instance and a connector instance attached to each other interact in a proper way. This is formalized by requiring that the functional behavior of the two instances, when projected on the interactions involved in the related attachments, is the same. The latter check,

instead, aims at guaranteeing that the actual parameters are consistent with the formal ones in case of architectural type invocation. This is formalized by requiring that the actual parameters do not alter the functional semantics of the architectural type w.r.t. component and connector interactions, i.e. the functional behavior of the architectural type when projected on such interactions. Technically, both checks are carried out by means of the weak bisimulation equivalence [6], a purely functional equivalence whose major feature is its ability to reason about the functional behavior of process terms when projected on certain actions, i.e. when abstracting from unimportant actions. It is worth observing that the architectural conformity checking can be compositionally conducted parameter by parameter as the weak bisimulation equivalence is a congruence w.r.t. the parallel composition operator.

We conclude by mentioning the fact that we are now in the process of implementing a software tool for the functional and performance analysis of well formed architectural types specified with the textual or graphical notation of ÆMPA. Such a tool will rely on the EMPA based software tool TwoTowers [2] and will allow us to conduct some case studies to assess the adequacy of our approach. For the time being, we are using a prototype of the tool to investigate the properties of several load distribution algorithms for replicated web services in different architectural scenarios.

Acknowledgements

This research has been funded by Progetto MURST Cofinanziato SALADIN: *"Architetture Software e Linguaggi per Coordinare Componenti Distribuite e Mobili"*.

References

[1] R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:312-249, 1997

[2] M. Bernardo, *"Theory and Application of Extended Markovian Process Algebra"*, Ph.D. Thesis, University of Bologna (Italy), 1999 (`http://www.di.unito.it/~bernardo/`)

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures"*, to appear in Proc. of the *2nd ACM Int. Workshop on Software and Performance (WOSP 2000)*, ACM Press, Ottawa (Canada), 2000

[4] M. Bernardo, P. Ciancarini, L. Donatiello, *"On the Formalization of Architectural Types with Process Algebras"*, to appear in Proc. of the *8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)*, ACM Press, San Diego (CA), 2000

[5] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985

[6] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989

[7] D.E. Perry, A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[8] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discpline"*, Prentice Hall, 1996

[9] C.U. Smith, *"Performance Engineering of Software Systems"*, Addison-Wesley, 1990