Component models for dynamic reconfiguration

Simon Bliudze* Hélène Coullon, Thomas Ledoux† Ludovic Henrio‡
Eric Rutten§ Rabéa Ameur-Boulifa¶ Françoise Baude‖

In recent years, distributed software systems have faced a set of new challenges raised by new Internet-scale distributed systems and highly dynamic infrastructures. Indeed, new kind of highly dynamic applications such as smart- and self-adaptive applications, as well as dynamic paradigms such as Fog-, Edge- or mobile-computing, and their associated dynamic infrastructures have entered the landscape relatively quickly without having programming support mature enough to meet safety, reliability and software-engineering-related properties. In this context dynamic reconfiguration management is gaining a lot of interest in many areas including languages, model-driven engineering as well as distributed systems. In this paper, we examine how component-based models can meet the challenges of dynamic reconfiguration.*

1 Context

1.1 Software Components

Modern software systems are inherently concurrent. They consist of components running simultaneously and sharing access to resources provided by the execution platform. These components interact through buses, shared memories and message buffers, leading to resource contention and potential deadlocks compromising mission- and safety-critical operations. Essentially, any software entity that goes beyond simply computing a certain function, necessarily has to interact and share resources with other such entities.

Software components have been designed to provide composition frameworks raising the level of abstraction compared to objects or modules. Components split the application programming into two phases: the writing of basic business code, and the composition phase, or assembly phase, consisting in plugging together instances of the basic component blocks. Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. Indeed, in component applications, dependencies are defined together with provided functionalities by the means of ports; this improves the program specification and thus its re-usability. Some component models and their implementations additionally keep a representation at runtime of the components structure and their dependencies. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities as this allows the component system to be reconfigured. For example the application can be adapted, e.g. to evolve in the execution environment, by changing some of the components taking part in the composition or changing the dependencies between the involved components. Reconfiguring a system consists in changing the configuration while a system is running. In particular, reconfiguring a system not only consists in changing the parameters configuring the system, but also changing the set of entities that constitute a system and their dependencies.

The term *software components* is sometimes taken with different acceptations, we want here to accept a very flexible definition of components: we can call components both very structured

*Inria Lille – Nord Europe
†IMT Atlantique, Inria, LS2N
‡Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
§Univ. Grenoble Alpes, Inria, CNRS, LIG
¶Institut Polytechnique Paris, Télécom Paris
‖Université Côte d'Azur

component systems (like CCM, Fractal [5], ...) and classical module systems, but also software packages, etc. All these composition systems are of interest, but depending on their structure, more or less adaptation capabilities, and more or less guarantees can be provided.

1.2 Distributed systems

In distributed systems, reconfiguration takes even more importance as the structure of components can also be used at runtime to discover services or adapt components in order to move them and execute them at a new location. Also the local configuration of the software entities takes a special meaning in a distributed settings as each software might need to be adapted (i.e., configured) to run on a new machine. In a distributed system, configuring a system consists in deploying this system, i.e. in placing each of the element of the system at a location, and configuring the hosting machine so that the software element can run properly. The placement of the entities is generally the solution of an optimisation problem, while the local configuration often consists in installing and configuring packages and modules, configuring the operating system, and sometimes running containers or virtual machines.

Component models are generally adapted for distributed systems but some of them have been designed specifically to address the challenges of distributed computing, like GCM [3].

2 Research directions

2.1 Formal methods for safe components

Distributed computer systems are by nature heterogeneous and large in scale. Their behavior depends on the interaction of multiple software components on varied hardware configurations, making them complex systems that are difficult to fully understand. The possibility to execute actions in parallel is also one of the main reasons to use a distributed system, but it further adds to their complexity. As a consequence, the process of configuring, deploying, and reconfiguring these systems is prone to errors that may result in the system entering an incorrect state, and ultimately to loss of service. Diagnosing the cause of these errors is often difficult and time-consuming, adding to their severity.

Testing is often inadequate for this type of system, as the nature of the components and their composition is not always known during development. Even when those elements are fixed, errors often depend on the timing of interactions between components, or on specific workload and network conditions, and are thus unlikely to be discovered by testing.

To ensure the correctness of component systems, a more promising approach relies on *formal methods* that offer strong generic guarantees about the systems. These methods are based on an abstraction of the system, which is checked against an agreed upon specification given in a formal language. The granularity of the abstraction may range from a very coarse model of the system – useful to describe matters of interoperability between components – to a much more detailed semantic model for a given programming language, used to characterize the execution of a specific component or application.

Since domain experts are not always available to provide behavioural models of components, methods and tools are required that would extract these (semi-)automatically.

In this domain, we identify the following challenges:

- the design of modelling frameworks incorporating various paradigms but capable of sufficient detail to describe specific properties of a model or a given application and allowing one to reason on both the runtime component behaviour and the software structure;
- the design of tools based on formal methods for proving properties on such abstract models;
- (semi-)automatic generation of models from the component source or binary code;

- the design of integrated higher-abstraction level formal tools to help in the design of safe, fault tolerant and efficient reconfiguration of distributed software [7, 11, 14, 9, 2].
- Furthermore, traditional solutions that are practiced today for modelling distributed systems impose static structure and partitioning [1]. To support dynamic system behaviours, we need a theoretical foundation for systems modelling and analysis to deal with the potential dynamic reconfiguration. The theory and the tools should also integrate multiple system aspects including robustness, safety, and security.

2.2 Safe autonomic components

The ability to reconfigure components at runtime requires design methods in order to construct managers for the decision and control of when to reconfigure, and towards what next configurations. Such self-adaptation managers are the object of Autonomic Computing [10], where a feedback loop monitors evolutions and – based upon a representation of the managed system – takes decisions which are implemented by reconfiguration actions. Challenges concern:

- the design of components to be observable and controllable w.r.t. adaptation policies, which can be related to some activities in the GdR RSD;
- the design of run-time monitoring techniques to detect deviations between components and their models at run-time;
- the design of the decision and control, that can involve a variety of approaches, from rule-based programming to Machine Learning, or models and techniques from Control theory [13], which can be related to some activities in the GdR MACS;
- the decentralization of decision and control when facing large-scale geo-distributed infrastructures such as Fog and Edge computing for scalability reasons and to avoid single points of failure in the autonomic process (related to the GdR RSD);
- the design of a reliable reconfiguration control process, such that when model or architectural invariants are violated or when hardware crashes at runtime, recovery and rollback mechanisms can be applied. A transactional approach has been studied in the past [12] and should be generalised for any components models in the context of large scale distributed system.

2.3 Separation of concerns in reconfiguration

When designing a distributed software system, both in its development and management aspects, multiple actors are involved: (1) *developers* who are responsible for designing and coding a set of modules or components, and responsible for the design of their composition; (2) *sysadmins* who are system administrators responsible for upkeeping, configuring, and testing multi-users computer systems such as servers or Clouds; and (3) *end-users* of the distributed software system or application. As a reconfiguration is a runtime adaptation of a software system, modifications are dynamically applied and may impact one or multiple steps of the initial design. Thus, multiple actors are also involved in a reconfiguration process or in the design of an autonomous system. Enhancing the separation of concerns in the reconfiguration process is an important topic that raises, among others, the following challenges:

- the design of new reconfiguration-oriented programming paradigms such that reconfiguration interfaces and APIs are exposed to future other actors of the reconfiguration, and such that unpredicted reconfigurations could be handled later on;
- the design of reconfiguration models and languages at the DevOps level where sysadmins are faced to reconfiguration cases while not having much information on the internal behavior of each component to reconfigure nor their interactions [8, 6, 4] (related to the GdR RSD);

- the design of coordination models to handle multiple and possibly concurrent reconfiguration aspects (e.g. energy, security, etc.).
- the design of high-abstraction level languages and tools for the expression of QoS (Quality of Service) and QoE (Quality of Experiment) from the end-user viewpoint, and the associated translation to a coordinated and safe reconfiguration process driven by the different QoS/QoE aspects.

References

[1] R. Ameur-Boulifa, et al. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017. ISSN 2352-2208.

[2] T. Barros, et al. Model-checking Distributed Components: The Vercors Platform. *Electronic Notes in Theoretical Computer Science*, 182:3 – 16, 2007. ISSN 1571-0661. Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006).

[3] F. Baude, et al. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2), 2009.

[4] F. Boyer, et al. Poster: A Declarative Approach for Updating Distributed Microservices. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 392–393. 2018. ISSN 2574-1934.

[5] E. Bruneton, et al. An Open Component Model and Its Support in Java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054. 2004.

[6] M. Chardet, et al. Madeus: A formal deployment model. In *4PAD 2018 - 5th Intl Symp. on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018)*, pp. 1–8. Orléans, France, 2018.

[7] H. Coullon, C. Jard, D. Lime. Integrated Model-Checking for the Design of Safe and Efficient Distributed Software Commissioning. In *Integrated Formal Methods*, pp. 120–137. Springer International Publishing, Cham, 2019. ISBN 978-3-030-34968-4.

[8] R. Di Cosmo, et al. Aeolus: a component model for the Cloud. *Information and Computation*, pp. 100–121, 2014.

[9] L. Henrio, et al. Integrated Environment for Verifying and Running Distributed Components. In P. Stevens, A. Wasowski (eds.), *Fundamental Approaches to Software Engineering*, vol. 9633 of *FASE*. Perdita Stevens and Andrzej Wasowski, 2016.

[10] J. Kephart, D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[11] C. E. Killian, et al. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. ACM, 2007.

[12] M. Léger, T. Ledoux, T. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In L. Grunske, R. Reussner, F. Plasil (eds.), *13th international conference on Component-Based Software Engineering (CBSE'10)*, vol. 6092 of *LNCS*, pp. 74–92. Springer Berlin Heidelberg, 2010.

[13] M. Litoiu, et al. What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems? In *Software Engineering for Self-Adaptive Systems 3: Assurances*, vol. 9640 of *LNCS*. Springer, 2017.

[14] J. R. Wilcox, et al. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15. ACM, 2015.