# Safe and optimal component-based dynamic reconfiguration

Simon Bliudze[*] Hélène Coullon, Thomas Ledoux[†] Ludovic Henrio[‡]
Eric Rutten[§] Rabéa Ameur-Boulifa[¶] Françoise Baude[‖]
Adel Noureddine, Ernesto Exposito, Philippe Aniorte[**]

## 1 Context

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.

Although concurrency can arise in centralised systems, e.g. when several processes share a single processor, there is a broad spectrum of systems, such as Cloud, Fog, Edge, and cyber-physical systems (CPSs) that are distributed in nature. In addition to ensuring correctness of individual components, distributed systems must be correctly *configured*. Configuration consists in deploying the system, i.e. in placing each of its elements at a location, configuring the hosting machine so that the software element can run properly and establishing the connections among the system components. The placement of the entities is generally the solution of an optimization problem. The local configuration often involves installing and configuring packages and modules, configuring the operating system, and sometimes running containers or virtual machines.

Since the structure of components can be used at run time to discover services or adapt components in order to move and execute them at new locations, correct *dynamic reconfiguration* becomes an important aspect of the system execution. In particular, the local configuration of the software entities takes a special meaning as each component might need to be adapted, i.e. (re-)configured to run in a new environment.

**Software components** 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

[*]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
[**]Université de Pau et des Pays de l'Adour - E2S

basic component blocks. Component models provide a structured programming paradigm, and ensure re-usability of programs. Indeed, in component applications, dependencies are defined together with provided and required 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 run time of the components structure and their dependencies. Knowing how components are composed and being able to modify this composition at run time provides great adaptation capabilities as this allows the component system to be reconfigured. An application can be adapted to evolve in the execution environment by 1) adding or removing components, 2) changing some of the existing ones, 3) changing the connections among the components, or 4) reallocating components to different execution locations. System reconfiguration can involve any combination of these four kinds of adaptation.

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 very structured component systems (like CCM, Fractal, BIP, ...), 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.

**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.

Obtaining a manually written formal model of a system is often not realistic; this can be due to the necessary effort, the required expertise in formal methods, or simply because the domain expert is not available. As a consequence one of the major challenge to disseminate the use of formal methods to ensure safety of the component systems is the design of automatic or partially automatic methods for model extraction.

**Classification pattern and outline** The ability to reconfigure components at run time raises different scientific challenges that could be included in a global Autonomic pattern [15]. A widely-used approach to capture the different phases of a reconfiguration mechanism is the MAPE-K loop [1]: *monitor*, *analyze*, *plan* and *execute* the reconfiguration of a software system, while sharing a common *knowledge*. We adopt this pattern in the rest of this document to

classify the different challenges raised by the reconfiguration. We first discuss scientific challenges regarding the representation of knowledge about the software systems to reconfigure, including component-based models, formal models and ontologies with a goal to guarantee integrability and interoperability. Second, we introduce a few challenges that stand behind the M-A-P-E phases when managing a reconfiguration process, including control theory, machine learning as well as formal aspects of the management.

# 2 Knowledge representation: component-based models

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.

## 2.1 Formal models / formal methods for component models?

Several formal approaches exist for the verification of component models. They can be separated into different categories. First, the formal specification of component models themselves, these are very generic formalization that can be used to prove properties applicable to all component systems using a given component model. Second, formal tools allowing the specification of a particular component system, followed by the proof of correctness of its behaviour. Of course, this second category relies on a specification of the underlying component model but does not require this specification to be completely formalised: generic properties of a component model can be used as hypotheses for the proof of properties of component systems, especially when these proofs are (partially) automatised. Expanding the so-called *trust base*, by proving the properties of libraries used in component model implementations (e.g. execution engines) is not directly in the scope of this proposal but can serve as an opening for collaborations with other interest groups.

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 run-time 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, 16, 23, 12, 3].

Furthermore, traditional solutions that are practiced today for modelling distributed systems impose static structure and partitioning [2]. 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 Reference component models

Among component models, we distinguish *flat* models ($L^2C$, Madeus, Aeolus, actor models etc.) and *hierarchical* models (CCM, Fractal, SCA), such that components may themselves comprise components. Reconfiguration in a hierarchical model is often challenging, but when this is possible the compositional approach makes it easier to manage reconfiguration of large complex systems. Another distinguishing feature of component-based models is the way they treat composition. Most models enforce some discipline on composition. A typical solution is to rely on port-interconnection sometimes constrained by a type and/or a cardinality system (like in GCM). Lastly, models can be characterized by their means of describing the life-cycle of components, in other words their management. At the most basic level, components have only two possible states: they exist or not. Many models also distinguish between *active* and *inactive* components. To describe the life-cycle (management behavior) of components more precisely, other models allow the definition of an arbitrary number of states, and attach different possible states to each component with predefined or user-defined transitions between these states. Aeolus [9], Madeus [6] and Concerto [5] are component models enabling rich life-cycle definitions for instance. Concerning distributed systems, component models are generally adapted for distribution but some of them have been designed specifically to address the challenges of distributed computing, like GCM [4]. We identify the following challenges:

- Some component models handle the functional behavior of each component and by composition the functional behavior of the entire distributed software system. Others handle the management behavior by enhancing the life-cycle modeling. However, this life-cycle modeling is correlated to the functional behavior and can be considered as an abstraction of the functional behavior. The combination and interactions of these behavioral approaches has not been tackled yet.

- While several different component models exist, the classification given above is a quite broad categorization that should be refined and could then be better adapted;

- When this classification is mature enough, it will be very useful to revisit each of the existing formal approaches and show that they are applicable not only on a given component models, but more generally on a set of models that have some well-defined characteristics. This could allow reusing existing formal approaches outside the component model they were originally designed for.

## 2.3 Domain-specific component models and architectures

One of the key challenges preventing large-scale adoption of component models and, particularly, the approaches based on formal methods is the necessity for the designers to learn a new model or language. Designed choices made by the scientific community are most of the time driven by the quest for conceptual minimalism in order to ensure that the resulting frameworks are easy to reason about (e.g. prove their expressive power, build broadly applicable analysis tools). The resulting modelling frameworks powerful but abstract, making it difficult for developers to map business concepts specific to their application domain to modelling concepts of the component framework. This situation can be compared to programming in an assembly language, highlighting the necessity of designing appropriate high-level *domain-specific* modelling languages.

A major example of an application domain for software reconfiguration is that of Cyber-Physical Systems (CPSs). Designing and managing CPSs and Cyber-Physical Systems of Systems (CPSoS) requires the definition of new methodologies for Model-Based Systems Engineering (MBSE). In CPSs, the exponential growth of collected data adds an exponential challenge to analyze these data and produce new information and discover relevant knowledge [10]. Additionally, with the diversity of CPS and their coexistence in CPSoS, new reference architectures are needed (for example, new propositions are being made for Industry 4.0, such as RAMI 4.0 [14] or IIRA [18]).

Finally, CPS adds their share of complexity in managing non-functional requirements, such as performance, energy, security, reliability, privacy, quality of service, etc. Software systems of CPS need to autonomously guarantee certain constraints and properties. We identify three main challenges for non-functional requirements in CPS:

- Energy consumption: using software eco-design and autonomic management and reconfiguration, CPS energy consumption can be controlled and reduced. This can be achieved by leveraging the computing or memory of CPS to offload workloads from cloud environments into CPS. The goal is to have a holistic view of energy costs across CPS and its environment, and harness the unused resources of CPS.

- Security and reliability: we aim to improve the trust between the actors of CPS, the protection of the collected or generated data, and access control of individual actors in CPS and of CPS in CPSoS.

- Performance: the aim is to build a software architecture and data processing methodologies improving the performance of CPS and CPSoS (such as optimizing productivity in an industry 4.0 environment, which might require balancing between actors with various levels of performance). Also, the scaling of CPS adds an additional architectural challenge: with CPS, the scale is much larger with a huge number of actors, interactions, data collection and data processing, all of which need to be managed and reconfigured autonomously.

Although the above challenges are formulated for CPSs, many other domain-specific component models and reference architectures exist for a broad range of domains: automotive [11], avionics [21], space [13], cloud applications [20, 24] etc.

### 2.4 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 run-time 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 unpredictable reconfiguration 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 [5, 9] (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.

# 3 Automatic reconfiguration: The MAPE approach

In this section, we divide the open issues related to the different phases M(onitoring), (A)nalysis-(P)lanning and (E)xecute.

## 3.1 Interfaces for monitoring and execution of reconfiguration

One the one hand, to handle autonomic reconfiguration the monitoring phase offers ways to introspect the software system to reconfigure, and the environment in which it evolves, including the infrastructure that host pieces of software. Execution of the reconfiguration, on the other hand, offers ways to act on the software system and its environment according to the decision taken by the analysis and the planning phases (see next section). Identified challenges are:

- 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 efficiency of the reconfiguration execution such that disruption time of services and pending time while the new configuration is available are minimized [5, 6];
- formal guarantees of the attainability of a reconfiguration execution [7], its applicability to the current system, it robustness etc.

## 3.2 Analysis, planning and control of reconfiguration

The analysis and the planning phases of MAPE are the ones holding reconfiguration decisions. Indeed, the analysis decides among a set of possible new configuration the new one and the planning synthesizes from the current configuration a plan to reach the new configuration. Both these phases are very challenging from optimization and verification viewpoints. Challenges are, among others:

- 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 [19], which can be related to some activities in the GdR MACS;
- the design of a reliable reconfiguration control process, such that when model or architectural invariants are violated or when hardware crashes at run time, recovery and rollback mechanisms can be applied. A transactional approach has been studied in the past [17] and should be generalized for any components models in the context of large scale distributed system.

### 3.3 Decentralization of MAPE

An additional aspect that should cover all phases of a reconfiguration process is its decentralization for scalability reasons and to avoid single points of failure in the autonomic process (related to the GdR RSD). At the monitoring level, the data management have to be handled in a decentralized way. The decision phases should also be decentralized which is particularly difficult as a reconfiguration process is a global and highly coordinated process [8, 22].

## 4 Conclusion

This document illustrates the importance and the return of an interest for component-based reconfiguration. This interest is justified by the growth of dynamic and geo-distributed software systems and infrastructures, related for instance to cyber-physical systems, IoT, fog and edge computing. Furthermore, this document also shows that many challenges are raised by reconfiguration, and are far from being solved by the research community. For this reason, we advocate reconfiguration to be one of the main focus of the software engineering community in the coming years.

## References


[1] An Architectural Blueprint for Autonomic Computing. Tech. rep., IBM, 2005.

[2] 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.

[3] 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).

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

[5] M. Chardet, H. Coullon, C. Perez. Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto. In *Proceedings 20Th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. IEE, To appear.

[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] F. A. d. Oliveira, T. Ledoux, R. Sharrock. A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 179–188. 2013.

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

[10] E. Exposito. Semantic-Driven Architecture for Autonomic Management of Cyber-Physical Systems (CPS) for Industry 4.0. In *International Conference on Model and Data Engineering*, pp. 5–17. Springer, 2019.

[11] S. Fürst, M. Bechter. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 215–217. 2016.

[12] 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.

[13] A. Jung, M. Panunzio, J.-L. Terraillon. On-board software reference architecture. Tech. Rep. TEC-SWE/09-289/AJ, SAVOIR Advisory Group, 2010.

[14] H. Kagermann, et al. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group.* Forschungsunion, 2013.

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

[16] 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.

[17] 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.

[18] S.-W. Lin, et al. The industrial internet of things volume G1: reference architecture. *Industrial Internet Consortium*, pp. 10–46, 2017.

[19] 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.

[20] OASIS Standard. Topology and orchestration specification for cloud applications. Version 1.0., 2013. URL `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html`. [accessed 07/04/2020].

[21] J. L. Tokar. A Comparison of Avionics Open System Architectures. *Ada Letters*, 36(2):22—-26, 2017. ISSN 1094-3641, doi:10.1145/3092893.3092897.

[22] D. Weyns, et al. On Patterns for Decentralized Control in Self-Adaptive Systems. In R. de Lemos, et al. (eds.), *Software Engineering for Self-Adaptive Systems II*, vol. 7475 of *Lecture Notes in Computer Science (LNCS)*, pp. 76–107. Springer, 2013.

[23] 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.

[24] F. Zalila, S. Challita, P. Merle. Model-driven cloud resource management with OCCIware. *Future Generation Computer Systems*, 99:260–277, 2019. ISSN 0167-739X, doi:10.1016/j.future.2019.04.015.